Vol. 6 No. 2

acm queue

architecting tomorrow's computing

March/April 2008

Association for Computing Machinery

Interview: Akeley and Hanrahan

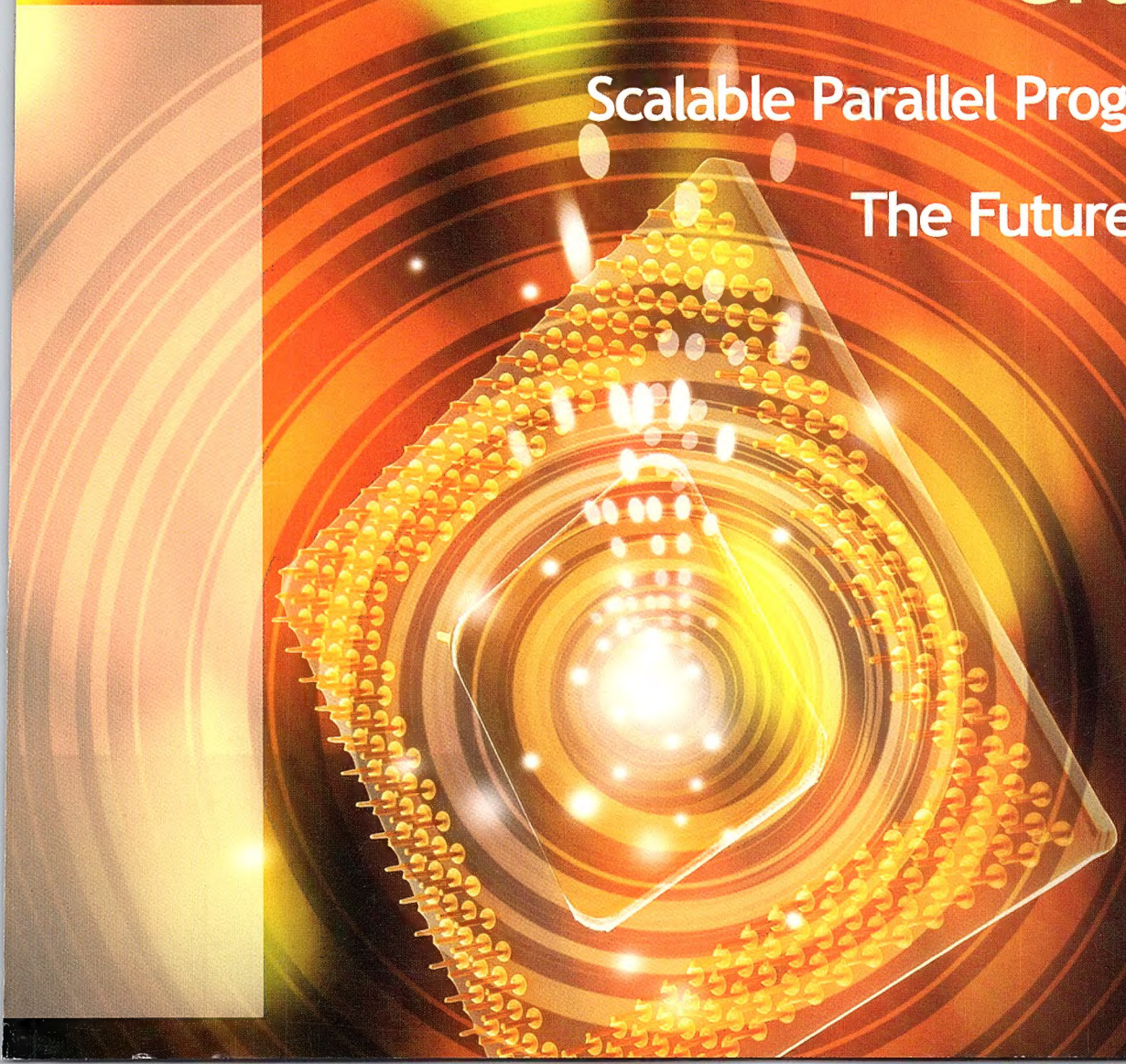
KV on Latency and Livelocks

GPUs

Not Just for Graphics

Scalable Parallel Programming

The Future of GPUs



www.acmqueue.com

_INFRASTRUCTURE LOG

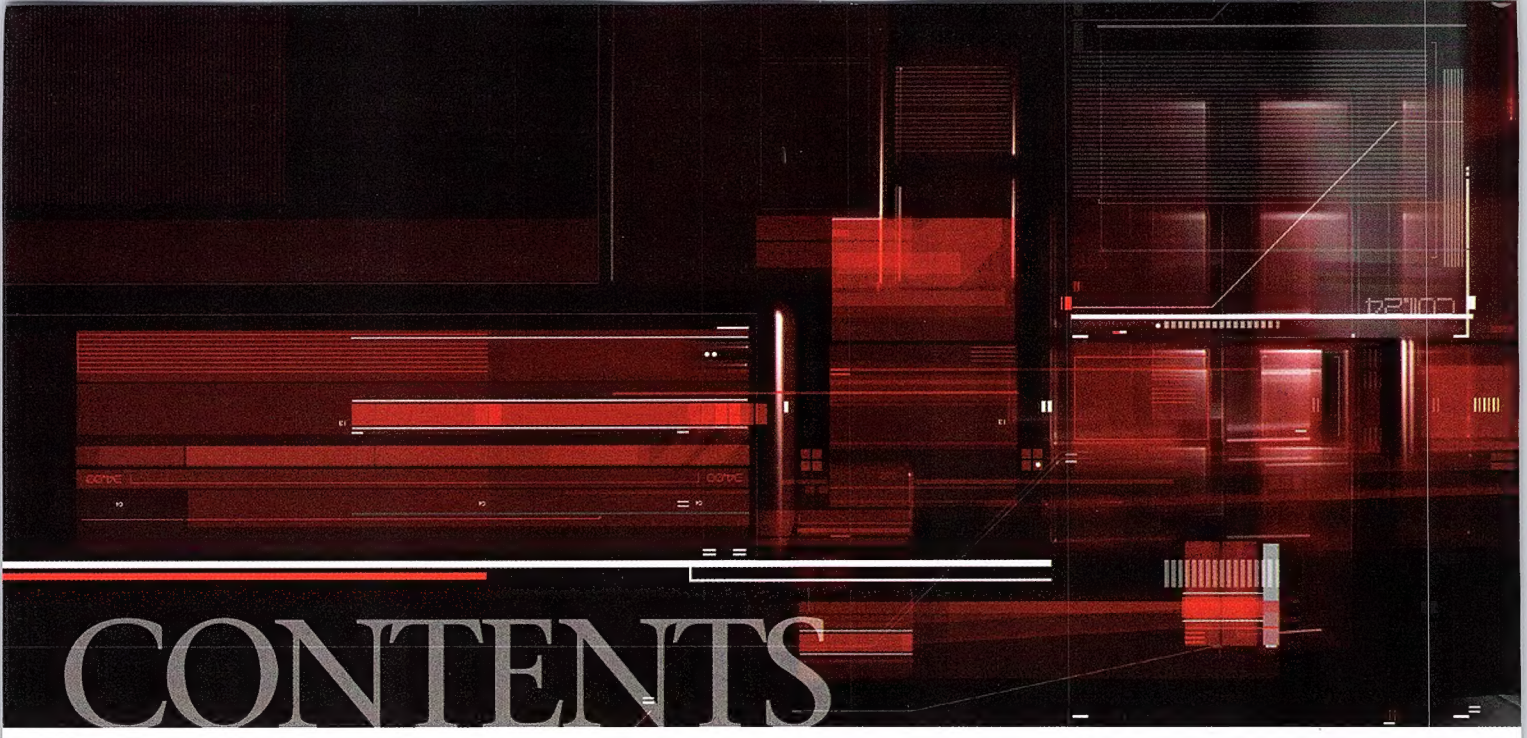
_DAY 72: We wrote our software but didn't build it to fit with the broader IT architecture requirements. Now we don't have the flexibility to reuse our assets. We're not moving forward. Why did we lock ourselves in like this?

_I never knew being stuck at work could be so literal.

_DAY 73: Here's something less confining. IBM Rational unifies all aspects of our SOA software design and development. Now we can ensure global architecture integrity using a new, simpler, modular systems development approach. And we're speeding our results with sound architectural design and automated service delivery and maintenance.

_I'm glad we're free. Was never sure where to put "stuck to my coffee cup" on my time sheet.

IBM.COM/**TAKEBACKCONTROL**/INNOVATE

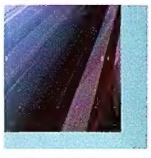


CONTENTS

MARCH/APRIL 2008 VOL. 6 NO. 2

FOCUS

GPUs



Your best source for software development tools!

programmer's paradise

LEADTOOLS Raster Imaging Pro

by LEAD Technologies

Raster Imaging Pro gives developers the tools to create powerful imaging applications. LEADTOOLS libraries extend the imaging support of the .NET framework by providing comprehensive support for image file formats (150+), 200 image processing filters, compression, TWAIN scanning, high-speed image display, color conversion, screen capture, special effects and more.

- .NET, API & C++ Class Library
- New Web Forms Control
- New Class Libraries for .NET
- Royalty Free

Paradise # L05 01101A02

$800.99

programmers.com/lead

dtSearch Engine for Win & .NET

Add dtSearch's "blazing speeds" (CRN Test Center) searching and file format support

New 64-bit Version!

- dozens of full-text and fielded data search options
- file parsers/converters for hit-highlighted display of all popular file types
- Spider supports dynamic and static web data; highlights hits with links, images, etc. intact
- API supports .NET, C++, Java, SQL and more; new .NET Spider API

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second." —InfoWorld

Single Server Paradise # D29072P

$873.99

programmers.com/dtsearch

VMware Infrastructure Acceleration Kits

New Release!

VMware Infrastructure 3 offers SMB organizations a scalable and cost-effective way to optimize utilization of technology assets, simplify IT management and protect the data and IT environments that run their businesses. Three new Acceleration Kits are now available, one for each of the three editions of VI3. These new kits offer midsize and smaller organizations and branch offices a cost-effective way to deploy a comprehensive virtualization solution that includes centralized management functionality. The ideal organization has a growing IT environment with between 15–60 servers.

Call for pricing on the Standard High Availability Acceleration Kit and Midsize Acceleration Kit.

Foundation Acceleration Kit Paradise # V55 47101A01

$2,662.99

programmers.com/vmware

DevTrack Small Team Edition

Powerful Defect and Project Tracking

by TechExcel

TechExcel DevTrack is the most powerful, affordable and easy-to-use defect and project tracking tool for development organizations. You'll dramatically transform your development processes, save significant time and resources, and deliver quality products on-time and on-budget.

- Sophisticated workflow engine
- Point-and-click administration
- Fully configurable user interface

5-User Pack Paradise # T34 0208

$1,414.99

programmers.com/techexcel

c-tree Plus®

by FairCom

With unparalleled performance and sophistication, c-tree Plus gives developers absolute control over their data management needs. Commercial developers use c-tree Plus for a wide variety of embedded, vertical market, and enterprise-wide database applications. Use any one or a combination of our flexible APIs including low-level and ISAM C APIs, simplified C and C++ database APIs, SQL, ODBC, or JDBC. c-tree Plus can be used to develop single-user and multi-user non-server applications or client-side application for FairCom's robust database server —the c-treeSQL™ Server. Windows to Mac to Unix all in one package.

64-bit SQL Available!

Paradise # F010131

$711.99

programmers.com/faircom

TX Text Control 14

Word Processing Components

New Release!

TX Text Control is royalty-free, robust and powerful word processing software in reusable component form.

- .NET WinForms control for VB.NET and C#
- ActiveX for VB6, Delphi, VBScript/HTML, ASP
- File formats DOCX, DOC, RTF, HTML, XML, TXT
- PDF export without additional 3rd party tools or printer drivers
- Nested tables, headers & footers, text frames, bullets, numbered lists, multiple undo/redo, sections, merge fields
- Ready-to-use toolbars and dialog boxes

Professional Edition Paradise # T79 02101A01

$811.99

Download a demo today.

programmers.com/theimagingsource

NEW: IP*Works! Version 8

by /n software

New Release!

The latest evolution of the most comprehensive suite of Internet communications components for professional developers is here!

A leap forward in design, performance, and new functionality with support for every major Internet protocol including - FTP, HTTP, SMTP, POP, IMAP, LDAP, DNS, RSS, SMS, Jabber, SOAP, WebDav, REST, ATOM, RAS, XML, and many more!

Call for pricing on the Java Edition

.NET Edition Paradise # D77 01201A01

$476.99

programmers.com/nsoftware

Vizioncore vReplicator

by Vizioncore

vReplicator is the real-time replication solution for the VMware ESX Server environment. Replication is performed outside the guest at the Service Console. The replication scheme is based on time elapsed and/or the size threshold for changes in the cache of the files being replicated (.VMDK and .VMX). With vReplicator, the entire virtual machine is replicated, including configuration settings, patches to the OS, the applications themselves as well as the data and all other OS-level changes.

Paradise # V79 04201E01

$441.99

programmers.com/vizioncore

Altova® MapForce® 2008

Visual Data Conversion, Transformation, and Integration Tool

by Altova

ALTOVA® mapforce®

MapForce: The premier data mapping, conversion, and integration tool from the creators of XMLSpy®. Through its visual interface, users can map seamlessly between any combination of XML, database, flat file, EDI, and/or Web service, then convert data instantly or auto-generate an application for recurrent transformations. Languages for code generation include: XSLT 1.0/2.0, XQuery, Java, C++, and C#.

New Release!

Enterprise Edition 1 user Paradise # IOD 03101A02

$1,184.99

programmers.com/altova

StorageCraft ShadowProtect IT Edition v3.0

by StorageCraft

Create, edit or restore backup images on as many servers, desktops and laptops as needed. Create online or cold state backups in minutes, no software installation required. StorageCraft™ ShadowProtect IT Edition provides complete bare metal recovery in minutes. ShadowProtect IT Edition provides IT Professionals with a bootable Windows environment to create and restore compressed and encrypted backups, no software installation required.

Paradise # SC5 03101A01

$3,294.99

programmers.com/storagecraft

Telerik RadControls

by Telerik

Add grid, combo, editing, navigation and charting functionality to your AJAX and ASP.NET projects. RadControls for ASP.NET enhances your Web applications by adding AJAX functionality to your ASP.NET projects. The suite takes full advantage of the features included in Visual Studio 2005. RadControls for ASP.NET helps developers deliver feature-rich, standards-compliant (WAI-A, WCAG 1.0, XHTML 1.1) and cross-browser compatible Web applications, while significantly cutting their development time. RadControls for ASP.NET includes: RadEditor, RadTabstrip, RadInput, RadCalendar, RadUpload, RadWindow, RadAjax, RadGrid, RadCombobox, RadMenu, RadSpell, RadChart, RadTreeview and more.

Single Developer Paradise # TB3 01101A01

$676.99

programmers.com/telerik

Multi-Edit 2006

by Multi Edit Software

Speed, depth, and uncompromising access to the inner workings of the machine, Multi-Edit 2006 delivers it all. A top tier program editor, ME2006 provides a single environment which can control all your VCS programs and compilers, and at the same time integrate with your existing RAD environments. Easily handle large (the only limit is your hardware) DOS/Windows, UNIX, binary, and Macintosh files in over fifty programming languages. Right out of the box, ME2006 comes ready to roll handling large DOS/Windows, UNIX, binary and Macintosh files in over 50 programming languages including Ruby, XHTML and more.

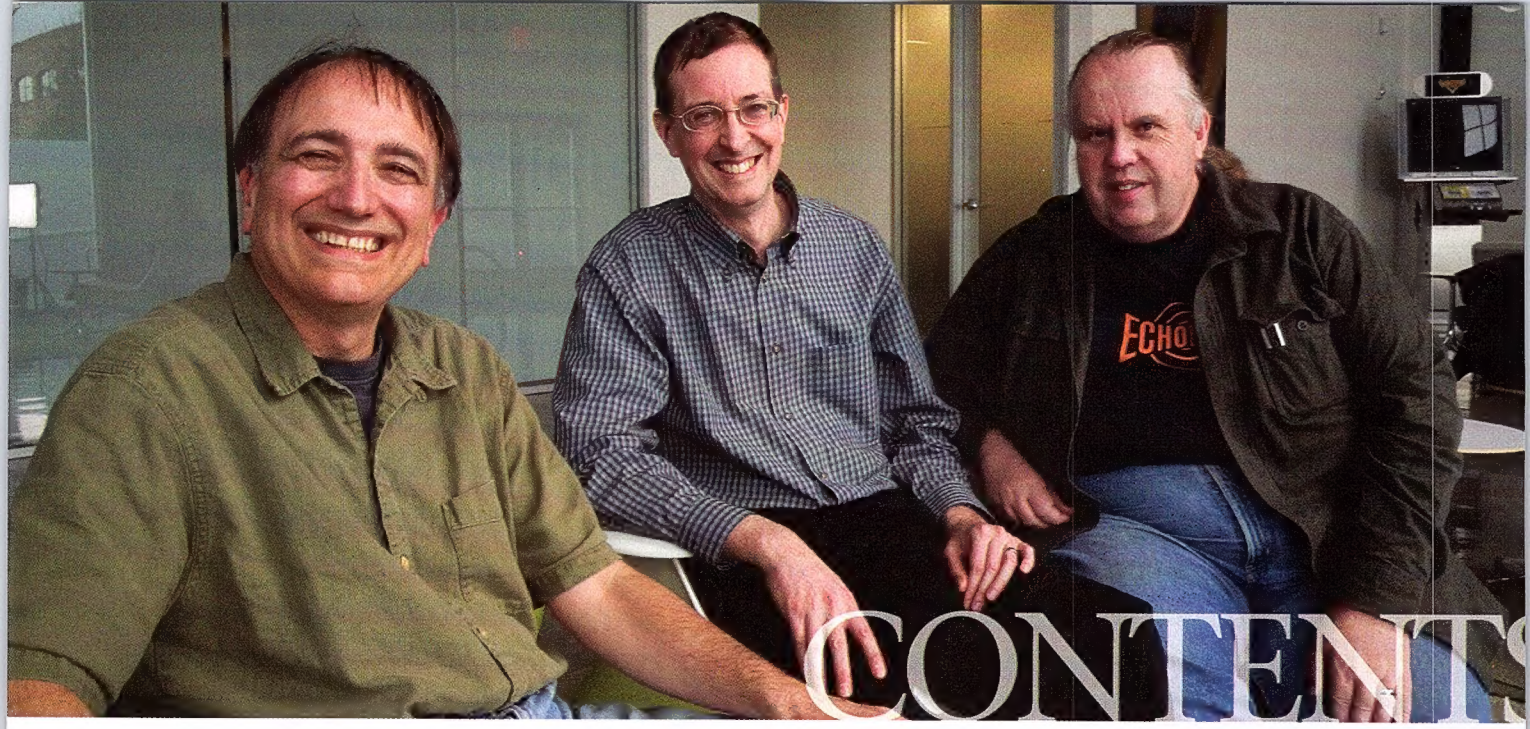
1-49 User Paradise # A30 01101A01

$87.99

programmers.com/multiedit

800-445-7899 **programmersparadise.com**

Prices subject to change. Not responsible for typographical errors.



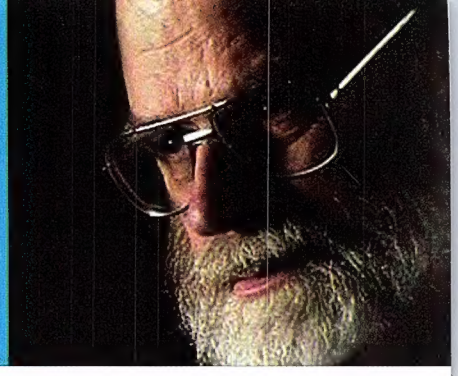
CONTENTS

INTERVIEW

DEPARTMENTS

Tribute to Honor Jim Gray

May 31, 2008
University of California, Berkeley



A Tribute Honoring Jim Gray:
Legendary computer science pioneer, known for his groundbreaking work as a programmer, database expert, engineer, and his caring contributions as a teacher and mentor.

General Session

Zellerbach Hall, UCB
9:00am – 10:30am

Speakers:

Shankar Sastry
Joe Hellerstein
Pauline Boss
Mike Olson
Paula Hawthorn
Mike Harrison
Pat Helland
Ed Lazowska
Mike Stonebraker
David Vaskevitch
Rick Rashid
Stuart Russell

All are welcome.
Registration is not required.

Technical Session

Wheeler Hall, UCB
Please see website for session times.

Presenters:

Bruce Lindsay
John Nauman
David DeWitt
Gordon Bell
Andreas Reuter
Tom Barclay
Alex Szalay
Curtis Wong
Ed Saade
Jim Bellingham

All are welcome.
Registration is required, see below.

Technical Session registration and additional information:

http://www.eecs.berkeley.edu/ipro/jimgraytribute

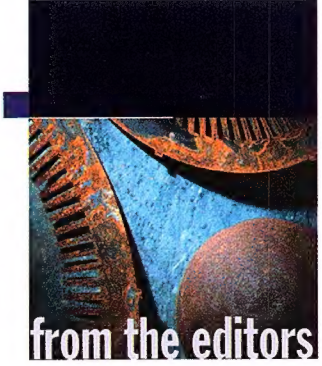
acm queue
architecting tomorrow's computing

Association for Computing Machinery

Advancing Computing as a Science & Profession

ACM Queue (ISSN 1542-7730) is published ten times per year by the ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701. POSTMASTER: Please send address changes to ACM Queue, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA Printed in the U.S.A.

The opinions expressed by ACM Queue authors are their own, and are not necessarily those of ACM or ACM Queue. Subscription information available online at www.acmqueue.com.



Going Digital

James Maurer, *ACM Queue*

Since its founding in March 2003, *Queue* has addressed the informational needs of the software development community through its printed version and Web site. Each issue has been carefully planned by a working board of prominent computing professionals and guest experts who meet monthly to set the magazine's editorial agenda and suggest and enlist the most qualified—and authoritative—authors. Our *Queue* "team" has greatly benefited from your feedback over the past five years. You've helped us shape a unique magazine, and we thank you for your input and your loyalty.

In response to a growing demand to offer a richer, more robust online presence for *Queue's* readership and in response to a changing business environment for specialized print magazines, ACM has decided to migrate *Queue* to the Web. As of July 2008, *Queue* will expand its publication frequency to 10 issues per year and publish issues online using the most cutting-edge digital-editions technology available, as well as revamp the existing *Queue* Web site to provide an overall improved user experience.

Queue's new site will expand beyond the magazine's core content to offer additional features and new sources of quality content, while also enabling and encouraging communication among respected members of the software development community. *Queue's* digital edition will look just like the print edition you receive now and will offer a number of useful features for enhanced navigation, search, linking, and browsing. You can view *Queue* in its new digital format by visiting http://mags.acm.org/queue/current.

Is there a way to receive *Queue* articles in print?

Yes, as of July 2008 a selection of *Queue* articles will appear in the new Practice section of the *Communications of the ACM*, the flagship publication of the ACM, with a current circulation of more than 87,000. In an effort to make the *Communications of the ACM* more appealing to a broader readership, including software developers and other practitioners in the software industry, the magazine is undergoing a complete redesign and editorial restructuring, which is scheduled to debut this July.

ACM Queue

ENTERS A NEW ERA

If you are not already an ACM Member or *Communications* subscriber, there are two easy ways to receive it:

1. Join. ACM Professional Membership includes a complimentary subscription to *Communications of the ACM* magazine. (Note students can also receive a print version of *Communications* by selecting the appropriate student membership type.)

ACM Professional Membership offers a host of additional career-enhancing benefits including:

- Unlimited access to 2,500 online courses from SkillSoft
- Unlimited access to 1,100 online books from Safari (featuring a large selection from O'Reilly) and Books24x7
- Discounts on ACM SIG conference registration
- Full access to ACM's new Career & Job Center with hundreds of targeted job postings
- *TechNews* and *CareerNews* e-mail digests and *MemberNet* newsletter
- The ACM Guide with more than 1 million bibliographic citations
- A free acm.org e-mail forwarding address with high-quality Postini spam filtering

Learn more and join at http://www.acm.org/joinacm2.

2. Subscribe. *Communications of the ACM* is now available at a lower rate for individual subscribers: http://www.acm.org/addpubs.

Thank you for making *Queue* one of the most highly regarded resources in the computing industry. Your support is greatly valued as we continue to address the challenges of new and emerging technologies.

Have questions or feedback? We'd love to hear from you via e-mail at queue@acm.org.

Latency and Livelocks

A koder with attitude, KV ANSWERS YOUR QUESTIONS. MISS MANNERS HE AIN'T.

Sometimes data just doesn't travel as fast as it should. Sometimes a program appears to be running fine, but is quietly failing behind the scenes. If you've experienced these problems, you may have struggled for a while and then become baffled and/or tired. Kode Vicious knows your frustration, and this month serves up some instructive words on how to deal with both of these annoying problems. Fatigued or mystified by other quandaries? E-mail your problem to KV@acmqueue.com.

Dear KV,

My company has a very large database with all of our customer information. The database is replicated to several locations around the world to improve performance locally, so that when customers in Asia want to look at their data, they don't have to wait for it to come from the United States, where my company is based.

A few months ago the company upgraded its software, which required all of the records in the customer database to be updated as well. When we tested the upgrade program it took only a few minutes to update a large number of records, but when we had to update the Asian customers, a process that had to run in Asia and that touched data in the U.S., the process began to take a lot longer. Since the company has a very fast network connecting the U.S. and the Asian offices, it's hard to understand how the distance could matter. There must be another reason for the time it is taking to run these programs.

Baffled with Bandwidth

Dear Baffled,

There is a big difference between having a big pipe and knowing how to use it. Two things matter in networking: bandwidth and latency. Unfortunately, most people think only of the former, not the latter. Latency is the time a message takes to get from point A to point B (e.g., a client and a server). If I were to guess, I would figure that you tested the conversion program on a local network, probably 100-Mbit Ethernet, where latencies are generally less than 1 ms. You then ran the program remotely across your very fast network and found that it ran much more slowly.

I bet it ran 100 times as slow. How did I pick 100 times?

Easy, the average round-trip time across the Pacific is about 100 ms. What you forgot was a very important constant, and then you forgot how networks work.

The very fast network you speak of was probably sold on a bit-per-second basis, so maybe you have a 1-Gbps link between Japan and the U.S., but that's a measure of bandwidth, not latency. It's the latency that matters in your case. Why? Because your conversion process very likely takes one record at a time, packs it up, makes a request to the server, and then waits for a response.

When the underlying network has very low latency, like a local network, this packing up of a single request and waiting for a response is barely noticed; but if you move to a higher-latency network, it crushes your system's performance under its iron heel.

The thing you forgot is *c*, the speed of light. Let's say you were running the conversion process in Tokyo and it was talking to a database in California. It's about 5,000 miles from Tokyo to California, and the speed of light is 186,000 miles per second, so a beam of light should be able to make it from Tokyo to California in about 0.027 seconds. That's 2.7 ms for the absolute fastest time between those two points, a round trip of 5.4 ms. That's already a factor of five slower than your LAN.

Of course, packets don't travel point to point. They are stored and forwarded at various points along their journey—that's how the Internet works. Each waypoint (the technical term is *router*) introduces its own bit of delay to the packet's journey, until what we have is an average of 50 ms each way between Japan and California. Now you have a difference of a factor of 100 between your LAN and the real network. Reality bites, and in this case it bit

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue *coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!*

2008 USENIX ANNUAL TECHNICAL CONFERENCE

June 22–27, 2008

BOSTON, MA

Join us in Boston, MA, June 22–27, 2008, for the 2008 USENIX Annual Technical Conference. USENIX Annual Tech has always been the place to present groundbreaking research and cutting-edge practices in a wide variety of technologies and environments. USENIX '08 will be no exception.

USENIX '08 will feature:

- An extensive Training Program, coverin0g crucial topics and led by highly respected instructors
- Technical Sessions, featuring the Refereed Papers Track, Invited Talks, Guru Is In Sessions, and a Poster Session
- Plus workshops, BoFs, and more!

Join the community of programmers, developers, and systems professionals in sharing solutions and fresh ideas.

www.usenix.org/usenix08/acq

USENIX

you hard. If converting some number of records took five minutes locally, it's going to take 500 minutes (8 1/3 hours) remotely. If I were you, I would bring a book to work, or download a movie like the rest of your coworkers, before you start any more database conversions.

There are ways to ameliorate these problems, though getting around the speed of light has eluded better minds for quite a while. The first is to run all the conversions locally; the second is to write the conversion program so that it batches requests. This makes more efficient use of the high-bandwidth network that your company has probably paid a small fortune to rent. If the database conversion on the server side can process a batch of records in less time than it takes to move all those records across the link, then you will gain some time; but if each record must be processed serially, then you're always going to suffer with slow performance over a long-distance link.

KV

Dear KV,

I've been trying to debug a problem in a network server. Under load the server seems to lock up, but when I look at the process status on the system, the state is always RUN, which means the process isn't blocked. The program is not in an infinite loop, because each time I attach a debugger, the code is in a different part of its processing loop, but there is never any output or progress. How can a program be running, not stuck in an infinite loop, and yet produce no output?

Dead Tired from Looking for a Deadlock

Dear Dead,

What you're looking at—actually what is staring you in the face—is a livelock, not a deadlock. Though most people learn about deadlock when they study computer science, they rarely learn about livelock until they're in one. Livelock is common in software where the general flow of the code is: read, process, write, read, process, write.

In a livelock situation the code is overwhelmed by the incoming load of work. Many systems—including, it would seem, your software—have a safety valve so that if overwhelmed, they can drop unfinished work. The problem is that the safety valve may not be good enough.

If a server drops a request because it runs out of time, memory, or other resources, then it will get stuck in a loop where it reads a request, tries to process it, and fails, then goes back and reads the next request, tries to process it, and fails. For all intents and purposes the code looks like it's running, and it is, but it's not succeeding at doing what it should. It's actually flailing and failing.

One way to overcome livelock is to buy more resources, such as faster processors or more memory. This "throw money at the problem" solution is intellectually cowardly and should be kept as a last resort. In many situations it's just not possible to get more raw hardware power since many companies already run their servers with the fastest processors and the maximum amount of memory. Usually companies do this because they have hired the wrong people to write their software (i.e., engineers who don't think about what resources their code is using—you know, idiots). For the nonidiots there are a couple of other ways to deal with livelock.

A common trick is to build code into the system that allows someone to tune the processing loop. The tuning is fairly simple: it says that the system will process only so many pieces of work per unit of time. Initially the knob is set to infinite, until the system winds up in livelock, and then the knob is turned down to just under the number of requests that caused the livelock. Yes, this means that the system will throw away requests for work, but it will still be able to do work, instead of being swamped and unable to do any work at all. Using this trick requires your software to know how many requests it's processing per unit of time, so you'll need some code to count that.

To implement a system in which the number of requests can be tuned, you need to design the code so that it can throw away requests as early as possible in the processing loop, perhaps even as far back as the read. If the system knows that it's overloaded, it should not waste additional resources trying to bring in any more work.

Often the problem in a livelock situation is related to processing time, in that there isn't enough of it. If you think that time is your problem, then you need to profile your software and see where it's wasting, er, spending its time. If the processing can be sped up, then obviously your code will be able to stay out of livelock longer.

Tuning software is a hard problem, and one that has been covered in *Queue* before (February 2006). Since I'm sure you've kept all your back issues, you should easily be able to go back and reference the excellent advice therein.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who makes San Francisco his home.

A Conversation with Kurt Akeley and Pat Hanrahan

Photography by Tom Upton

Graphics veterans DEBATE THE EVOLUTION OF THE GPU

Interviewing either Kurt Akeley or Pat Hanrahan for this month's special report on GPUs would have been a great opportunity, so needless to say we were delighted when *both* of these graphics-programming veterans agreed to participate.

Akeley was part of the founding Silicon Graphics team in 1982 and worked there for almost 20 years, during which he led the development of several high-end graphics systems, including GTX, VGX, and RealityEngine. He's also known for his pioneering work on OpenGL, the industry-standard programming interface for high-performance graphics hardware. Akeley is now a principal researcher at Microsoft Research Silicon Valley, where he works on cutting-edge projects in graphics system architecture, high-performance computing, and display design.

Hanrahan also has years of experience in computer graphics, including industry positions at Pixar and DEC and academic posts at Princeton and Stanford, where he currently teaches and does research. While at Pixar, Hanrahan helped design the RenderMan Interface Specification, integral in creating the sleek, photorealistic images

seen in Pixar's feature films. In 2004 Hanrahan received an Oscar for Technical Achievement in Computer Graphics for his role in modeling the ways light scatters below translucent surfaces such as skin.

Tom Duff, who works at Pixar, conducted our interview with Akeley and Hanrahan at Pixar's studios in Emeryville, California. A graphics-computing veteran himself, Duff has spent more than 30 years writing software for feature films, from *Star Trek II* to *Ratatouille*. His current passion is building robots and their software for theme parks and other entertainment uses. Duff has a lengthy resume of publications and patents and has received two Academy Awards for his scientific and technical achievements.

TOM DUFF What are GPUs and how have they evolved over the years?

PAT HANRAHAN Graphics requires a lot of computation. People who want realtime graphics want as many cycles as they can get as cheaply as they can get them. GPUs were developed in response to the demand for creating a lot of inexpensive cycles that you can use for graphics.

KURT AKELEY It used to be that there was so little compute power available for the amount of money somebody could spend on a desktop, that we really couldn't do very high-quality graphics at all. What that meant was the problem had to be reduced to something simple enough that you could build hardware that was very specific and solved a very specific set of problems.

The graphics hardware pipeline, the abstraction that shows up in some of the articles in this issue of *Queue*, was an answer to that: an architecture for solving a very small subset of what's required to do movie-quality graphics, but solving it efficiently enough that you could get it to work interactively 30 years ago.

What's happened is that over those 30 years, the Moore's law increase in transistor counts by 50 percent a year or so, and clock increases by maybe 20 percent a year over most of that period, resulted in a fantastic amount of computing power. This means the GPUs are actually solving a much more general problem now. There's enough compute power that you don't have to do just a little bit to a vertex and barely compute a pixel and stick it in a frame buffer—that was the whole story 30 years ago. Now you can do really advanced shading. Of course, that involves much more general operations.

So, where GPUs used to be quite specific and very distinct from CPUs, today we're having this collision in terms of architecture; what a GPU is and what a CPU is are no longer disjoint sets.

What we're talking about isn't just whether we can use graphics processors to do general-purpose computing, but in the bigger sense, how will general-purpose computing be done? How will graphics processing and other technologies that have evolved influence the way computing is done in general? That's a big issue that the world's going to be working through for the next five or ten years.

PH One way to think about it is that GPU architects, because they had this huge problem to solve—that is, making pictures in realtime—had to come up with some very innovative techniques to develop multicore chips. In the process of innovating, they've actually created things that are more general than they might have thought. I mean, they knew they were general, but now people are starting to discover them, or at least the ideas behind them.

Several years ago the major CPU chip vendors weren't interested in parallel computers. They would just say, "Clock rates will continue to increase; don't worry about parallelism." But then they decided a couple of years ago that they can't keep making these things faster and faster without using parallelism. Now everybody realizes that converting your programs to run on multicores is a big thing, and you have to do it or you won't get more performance.

You can view GPUs as a couple of steps ahead of the game. They were out there maybe a little bit further, and they still are out there further than the dual-core or quad-core CPUs. GPUs will have 16 or 32 cores, and they're specialized for certain different classes of workloads that are more related to graphics.

It's not that GPUs are on a weird, parallel track trying to solve only the graphics problem—they actually got ahead of the more-general computing game by innovating in computer architecture. That's very interesting, and if you're a programmer, it's the main reason you should be aware of these techniques and what's going on.

TD The interesting thing that's been coming down the pike for the past several years is using these processors for computational purposes that don't really have anything intrinsically to do with graphics. There were two competing directions driving all of this.

On the one side are the engineering workstations that SGI was building in the beginning that were running at very high speeds, basically just drawing lots of polygons with simple shading—a very circumscribed sort of thing. Pulling the other way is the trend toward using a very general model to describe shading.

Now, those things pull in opposite directions. The performance of old-school GPUs really depended on the fact that we knew exactly what the algorithm was. All of the control junk that was in a normal CPU was pretty much irrelevant.

KA It has been a smoother transition. People often say that programmability is a recent innovation in GPUs. Well, GPUs have been programmable for about the entire time that they've been built. With most SGI machines,

if you opened one up and looked at what was actually in there—processing vertexes in particular, but for some machines, processing the fragments—it was a programmable engine. It's just that it was not programmable by *you*; it was programmable by *me*. From an architecture standpoint, that's a fairly subtle distinction. What we weren't doing was *selling* application development. It's a little like mobile phones now. In general, they're not extensible except by a very small set of people, so they appear unprogrammable.

All along, those SGI machines had microcode engines that were programmable; we just weren't exposing the programmability to the world. Frankly, part of the reason was that we didn't have control of those components.

We went out to the market and said, "You know, the Intel 860 is the best floating-point-per-dollar solution this time, so we'll put in one of those and build a microcode engine that runs it."

Then the next time, we would go out and say, "Mmm, this TI 40-bit floating-point gizmo is the best one, so we'll use that." We couldn't promise the same coding environment generation after generation, so we couldn't reveal that it was programmable or else our customers would get very upset. We tried that. It actually does upset customers when you let them invest in coding and then sell them another machine that's faster but doesn't run their code. So for a variety of sort of tactical reasons, the programmability wasn't exposed.

The story is more complicated now because there is less programmability in some areas. But the general notion that people woke up eight years ago and said, "Oh, it makes sense to put programmability in these things," is definitely oversimplifying. This architectural trend has been smoother than that.

TD There was, in fact—and I was here for this—an awful lot of resistance from the big players in the GPU business to exposing that programmability.

KA I was part of that.

TD I like to think that the transition happened because of us in the movie-quality imaging business. We pressed hard for it and demonstrated that if you were going to make high-quality images, this was the way you were going to do it.

PH They always knew you were right; it's just that it was too costly for them to consider. The market opportunity wasn't there. But the games eventually started getting so sophisticated that there was no way of making them look better without exposing programmability to the [John] Carmacks and [Tim] Sweeneys of the world.

KA Games were a big enough market that you could afford to do it. That's the part that's less obvious. It cost a huge amount of engineering, and it took a lot of steps and a lot of years to build this into the marketplace, which is bigger than movies at this point. It costs a lot of money to engineer these things, so it wasn't like you could just wake up one day and say we ought to do it. It took all these years to build up the capital expenditure capability that an Nvidia or an ATI has to actually do it.

If mistakes had been made along the way—big ones—it wouldn't have happened. There are lots of examples of marketplaces where there was custom hardware that hasn't beautifully evolved into the space the way graphics has. I think a lot of that is market opportunity; it's not pure technology. Those markets just wouldn't support it.

TD If you look at the big computing machines in the world, you see that most of them are devoted to fluid dynamics and electrostatic simulation, for sort of obvious defense-related reasons.

PH And *n*-body calculations. To me, graphics is mostly about simulation. There are basic computational building blocks that go into simulation. To the extent that graphics uses a certain set of those in certain ways, a lot of other people use other sets of those in other ways. Once you start seeing the building blocks designed for simulation in a fairly general-purpose parallel way, you can say, "Yeah, it's not just for graphics; it could be used for other things." That's what other people are starting to find out.

TD We've heard that GPU performance increases faster than Moore's law. Is that just low-hanging fruit because of the primitive state of GPU architectures, or is this trend going to continue? Are those CPU and GPU curves going to merge?

KA Moore's law, just to be clear, has to do with transistor count and is formulated, I think, as an economic law that the number of transistors on the most economically produced die size will go up exponentially—and it turned out around 50 percent a year has been the number. So 1.5 is the compound average growth. But remember, it isn't a performance law; it is a transistor-count law.

TD Sure, but performance is related.

KA Performance is related to both transistor count and clock speed, and the clock speed mattered a lot. The clock speed has been going up around 20 percent a year.

PH One way to think of it is sort of as a rate-cubed effect. You get a square for the area, and if something shrinks in size by a half, you get four times as many of them, but the clock also goes up by roughly a factor of two.

KA It's not purely linear, but if you go back and look at

the compound rates, you could argue that if performance is the total number of transistor transitions per second—that's a reasonable proxy—and call that capability, then that's the compound of the Moore's law transistor count and the clock-rate increase. Those two things together are the capability rate, and they have been going up roughly 1.7 to 1.8 per year, until the past few years, for quite a long time.

That's how much faster you would expect an idealized thing to get year over year if the people doing it weren't getting any smarter, if they weren't learning anything. Indeed, GPUs have been getting faster by some metrics—not all, but by some—at a rate a little bit faster than that capability rate. So, we can say that their designers have been getting smarter.

CPUs are intrinsically sequential, which means they have a single thread of execution. The transistors didn't go to more computation; they went to all kinds of cleverness to feed that one engine faster. It's an interesting historical quirk that for a while the increase worked out close enough to a 1.5 compound growth rate that people started calling that Moore's law, but it's not the same thing.

PH This is really important for people to realize. You had this potential for CPUs to go, say, 75 percent faster every year, but they got only 50 percent faster. That means they were losing 25 percent a year to what they could have achieved every year since the dawn of the microprocessor. Not only that, 25 percent of the 50 percent was for free because the clock got faster. So they had 50 percent more area or more transistors. They used only 25 percent of the capability of those extra transistors. Fewer than half of their extra transistors were turning into anything useful. That sounds like bad engineering to me.

When I used to consult at SGI, Kurt told me that if we turn only half of our new transistors into performance, we haven't done our job as engineers. Our goal as engineers is to use our resources fully. Since the dawn of the microprocessor, however, we've been throwing away half our transistors. That's just another way of saying how inefficient CPUs have become.

Now, when GPUs hit the market, they got a performance increase of about a factor of 20 over CPUs. One way of thinking about it is GPUs put us back on the Moore's law curve—not the number-of-transistors one, but the real capability curve. CPUs have never been on that curve.

KA And GPUs have arguably exceeded it, but when you look carefully at the numbers, the bandwidths aren't going up at those rates. There's more compression. Some trickery and clever engineering have made them get faster by a bit. Plus, the raw capability gives you this huge disparity between GPU peak performance on problems that are suited to them and what you can get on a CPU.

The interesting thing is that people in the CPU world are not sitting on their hands anymore. As soon as they made the decision to go parallel, the gloves came off. They're going to stop squandering all those transistors on trying to make one thread go incrementally faster, and they're going to start using them to make a bunch of threads go faster. This puts them potentially on the same curve as GPUs. One prediction you might make is that this disparity is going to stop changing so quickly as it has for the past 20 years or so.

TD When you get into this sort of architectural discussion, the first question that always has to come up is, where's the bottleneck? Here's where I see the problem right now: if I have a nice piece of silicon with 64 or 128 cores on it, and it's only got a few hundred or a thousand pins, there's still a serious communication problem off-

chip. We don't see much progress happening on that.

PH Right. Don't fool yourself that this problem will be solved.

TD Really? When Seymour Cray was building the fastest computers in the world, it was precisely by addressing that problem, by making memory buses that were enormously wide paths to memory.

PH Let me tell you why my intuition is that the problem won't go away. If you look at the cost of computing, it's about communication. That's where all the power goes. It's hard and expensive to provide that bandwidth. Assuming the most expensive part is usually well engineered, you try to do the best job you can with the parts of the system that matter. People are working as hard as they can at making communication costs lower. The low-hanging fruit is to take the problem away from being one involving communication to one that doesn't involve your most expensive resource.

Our programming environments have to be more aware of communication. Let's say every time you said "equal sign," you thought 1,000 times more power was being exerted than when you said "multiply."

Bill Dally [chair of the Stanford University computer science department] has this great number, just to put this in context. If you build a 32-bit floating-point unit, it takes a picojoule to do the floating-point operation. If you execute a 32-bit floating-point instruction on a processor, it takes a nanojoule, 1,000 times more power.

The actual computing part was free, but sending the data to the floating-point unit, reading it back, putting it in the cache, and trying to put it onto the bus uses 1,000 times more power. You're just fighting physics. Physics tells you communication is expensive, and your programming model has to revolve around the communication if it is going to be efficient. So, that problem is not going to go away—there's just no way to defeat physics.

KA The way to minimize communication is by coherence, by having like things happen in like space and like time. Parallel processors, SIMD (single instruction, multiple data), are just a way of establishing execution coherence; putting in cache memory is a way to create locality, but it's a very general way.

Again, the CPU people gave us a really pleasant abstraction. But in a C program, that equal sign might be a nanojoule or it might be a millijoule, depending on what actually happens. There's no visibility into that to a C programmer. It's really hard to look at a C program and detect that 1000:1 difference in the cost of that equality, an assignment operator.

On the other hand, in a parallel-programming environment—a fairly crude one today—it's quite visible to you because you're handed something that's data-parallel, and you deal with the fact that, roughly speaking, the same thing is happening to similar data all at the same time. By being willing to deal with that, you've been able to get this huge increase in coherence that allows the performance to happen for a reasonable amount of power or a reasonable amount of communication. So the question is, what are some abstractions we can find that aren't onerous to program to but that allow those things that matter to perform and to become more visible to programmers so that they can make more reasonable choices, or abstract them away so that choices are made automatically?

But, again, the fact that a modern CPU has so much of its die area dedicated to cache is expensive. It's saving power, but it costs a lot of die area and power to save the power. You can always do better if you move more responsibility to a higher level.

TD The idea, then, is moving the work to where the data is instead of moving the data to where the work is.

KA It's both. The important thing is having them be near each other. The original graphics pipeline was this gorgeous example of that: do a bunch of work here, move the data to something right next door and do a bunch more work, and then move it to something right next door and do a bunch more work.

If texture mapping hadn't come along, your argument that graphics systems would be worthless for general-purpose computing would be true. Texture mapping is this awful sort of incoherent thing. It has some coherence, but as you put it into a shader and allow people to generate texture addresses, eventually you can completely destroy the coherence.

TD It's incoherence, but it's a scatter/gather kind of incoherence.

KA It's a gather mostly, the way GPUs deal with it. The point is, as they've dealt with that more and more, the communications have gotten a lot richer. That ability to gather is a huge distinction from the old pipeline that really had no communication between the elements.

Dealing with that lower level of coherence has made the machine much more general purpose. It turns out that you can do that by caching a lot more cheaply than you can with the general-purpose caching on a CPU, so it's not all the way to that extreme. But it's a lot less coherent than the non-texture mapped pipelines that I started with. They were almost perfectly coherent.

TD Pat, a couple of years ago, one of your former students gave a talk here about the future of computing on GPUs.

His claim basically was that all of Pixar's fancy rendering stuff—ray tracing and subsurface scattering and more complicated simulation effects—doesn't happen on GPUs these days. He pointed out a series of papers that covered basically everything that we do that's really hard.

The conclusion he drew from that was there's no reason not to run the whole thing on a GPU right now, but the examples he showed us were all isolated examples that don't play together. It was pretty obvious that of these separate pieces, there was no reasonable way to build a whole system. They all required different data structures for storing geometry and dealing with piles of rays. I guess the point is that kernels are not systems.

PH Exactly.

TD Two things to consider: First, somebody needs to be thinking about how to bridge that. The system-integration problem is really hard.

Second, unless the architecture of GPUs evolves in ways that I don't expect, they're going to be attached processors forever and there's going to be a general-purpose processor somewhere that's doing some of the work. How the work is allocated between two different heterogeneous kinds of machines is a really important problem, and it's really hard because optimization strategies on the two kinds of machines are fundamentally different.

PH That is a great point, and I think that is actually the biggest challenge facing us right now—for another important reason, which we haven't talked about.

As you probably know, AMD acquired ATI and both Intel and AMD are working on building heterogeneous multicore systems that basically combine a CPU and GPU on a single chip. In the future, it might even have some other specialized hardware on it, such as a video codec. This will be our mainstream computing platform.

A laptop, for example, will have one of these single-chip things in it. How are we going to program this thing? How are we going to schedule work on it? How are we going to deal with different instruction sets or different vector units?

I don't really know, but I do know that people are going to build these things, and we had better start thinking about it. It's going to be very challenging to figure out.

KA One way to think about this is to figure out what we're going to mean by GPU and CPU over the next few years, and what is the difference between the two? A lot of people right now think of something that's data-parallel, with lots of execution units, as a GPU, and something more sequential as a CPU. But that's not going to be the right distinction down the road.

TD Certainly, a multicore Intel box with 64 or 128 CPUs on it looks an awful lot like a data-parallel machine from 50,000 feet.

KA But it has a fundamental difference, and I think in some underlying way this may get at your issue: ultimately, the way the resources are deployed and harnessed and the way the data is moved around on a CPU is under software control; the way the data and resources are deployed on a GPU is still significantly under non-software control.

There's a lot of general-purpose computing in there, but the way it's wired together, the way the data moves, is not general purpose or at least not exposed yet. It's still a graphics pipeline, or it's pretty much neutered in something like CUDA. You lose this notion of wiring a bunch of different things together, and you're pretty much given a single data-parallel space to operate in. I'm simplifying a bit here, but that's roughly true. In some sense, what makes something a GPU is that the resources aren't organized by your software control; they are organized by somebody else.

Think back to the old 860. It was an Intel part that had a general-purpose CPU, but it had a little rasterizer thing on the side. It's very clear that the CPU directed the rasterizer; the rasterizer didn't direct the CPU. If you open up a GPU, the rasterizer is pretty much what doles out the work that makes the high-performance thing go.

In some sense, it's that orientation that determines if it is a CPU or a GPU. When GPUs evolve to the point where that's no longer true, that's the day that some of your lower-level concerns get addressed. You say, "Gee, they've got different data structures, and how do we wire all this stuff up?"

Once you free up the special-purpose stuff to be slaves to the general-purpose stuff, instead of having the general purpose be a slave to the special purpose, that's what software programmers are used to. That's what allows you to change data structures and organize the shape of your overall computation.

I don't think GPUs are so far away from that, and when that threshold is crossed, then there really aren't GPUs and CPUs anymore. Now there are just resources that are optimized for highly parallel computation.

PH It's a neat way of thinking about it.

TD Yes, it is.

KA And it gives you a chance to flip your hands up. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

A gamer wanders through a virtual world rendered in near-cinematic detail. Seconds later, the screen fills with a 3D explosion, the result of unseen enemies hiding in physically accurate shadows. Disappointed, the user exits the game and returns to a computer desktop that exhibits the stylish 3D look-and-feel of a modern window manager. Both of these visual experiences require hundreds of gigaflops of computing performance, a demand met by the GPU (graphics processing unit) present in every consumer PC.

GPUs
a closer look

As the line between GPUs and CPUs begins to blur, it's important to understand what makes GPUs tick.

KAYVON FATAHALIAN
and MIKE HOUSTON,
STANFORD UNIVERSITY

GPUs
a closer look

The modern GPU is a versatile processor that constitutes an extreme but compelling point in the growing space of multicore parallel computing architectures. These platforms, which include GPUs, the STI Cell Broadband Engine, the Sun UltraSPARC T2, and, increasingly, multicore x86 systems from Intel and AMD, differentiate themselves from traditional CPU designs by prioritizing high-throughput processing of many parallel operations over the low-latency execution of a single task.

GPUs assemble a large collection of fixed-function and software-programmable processing resources. Impressive statistics, such as ALU (arithmetic logic unit) counts and peak floating-point rates often emerge during discussions of GPU design. Despite the inherently parallel nature of graphics, however, efficiently mapping common rendering algorithms onto GPU resources is extremely challenging.

The key to high performance lies in strategies that hardware components and their corresponding software interfaces use to keep GPU processing resources busy. GPU designs go to great lengths to obtain high efficiency, conveniently reducing the difficulty programmers face when programming graphics applications. As a result, GPUs deliver high performance and expose an expressive but simple programming interface. This interface remains largely devoid of explicit parallelism or asynchronous execution and has proven to be portable across vendor implementations and generations of GPU designs.

At a time when the shift toward throughput-oriented CPU platforms is prompting alarm about the complexity of parallel programming, understanding key ideas behind the success of GPU computing is valuable not only for developers targeting software for GPU execution, but also for informing the design of new architectures and programming systems for other domains. In this article, we dive under the hood of a modern GPU to look at why

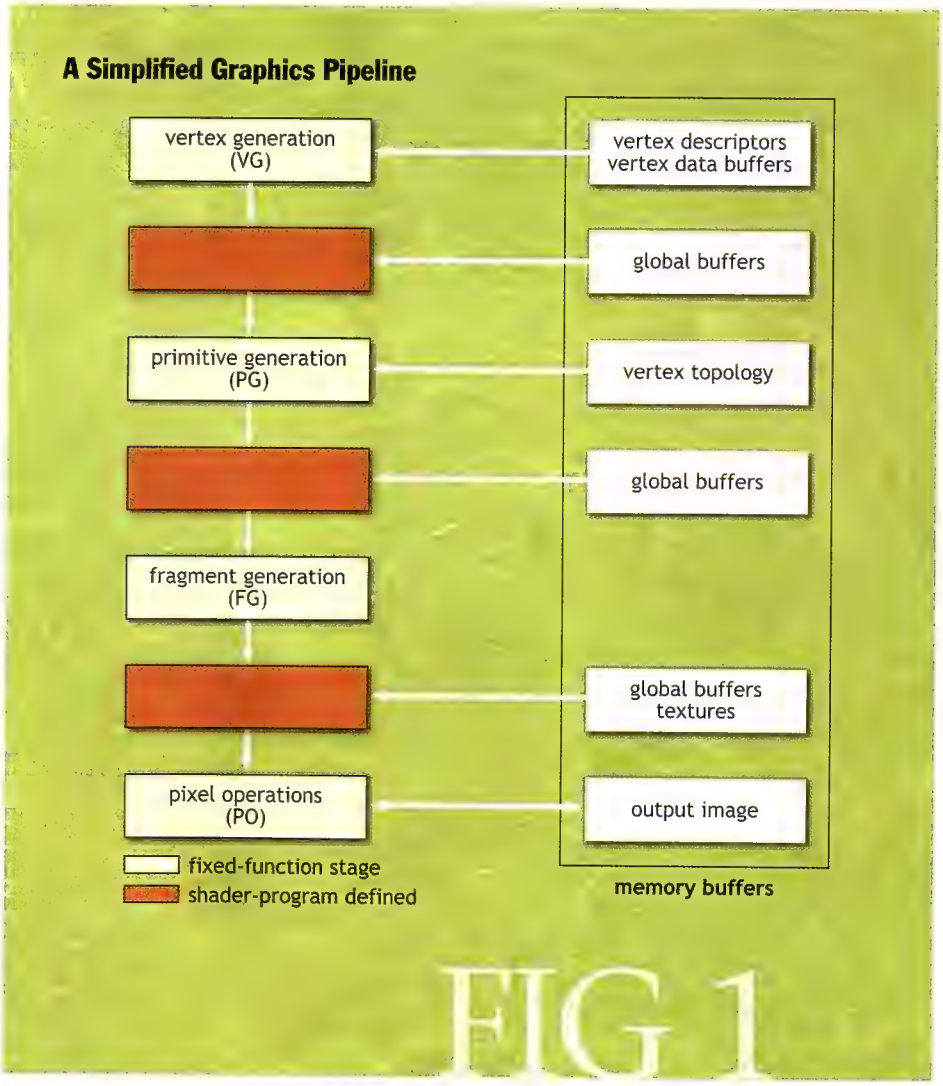


FIG 1

interactive rendering is challenging and to explore the solutions GPU architects have devised to meet these challenges.

THE GRAPHICS PIPELINE

A graphics system generates images that represent views of a virtual scene. This scene is defined by the geometry, orientation, and material properties of object surfaces and the position and characteristics of light sources. A scene view is described by the location of a virtual camera. Graphics systems seek to find the appropriate balance between conflicting goals of enabling maximum performance and maintaining an expressive but simple interface for describing graphics computations.

Realtime graphics APIs such as Direct3D and OpenGL strike this balance by representing the rendering computation as a *graphics processing pipeline* that performs operations on four fundamental entities: vertices, primitives, fragments, and pixels. Figure 1 provides a block diagram of a simplified seven-stage graphics pipeline. Data flows between stages in streams of entities. This pipeline contains fixed-function stages (tan) implementing API-specified operations and three programmable stages (brown) whose behavior is defined by application code. Figure 2 illustrates the operation of key pipeline stages.

VG (vertex generation). Realtime graphics APIs represent surfaces as collections of simple geometric primitives (points, lines, or triangles). Each primitive is defined by a set of vertices. To initiate rendering, the application provides the pipeline's VG stage with a list of vertex descriptors. From this list, VG prefetches vertex data from memory and constructs a stream of vertex data records for subsequent processing. In practice, each record contains the 3D (x,y,z) scene position of the vertex plus additional application-defined parameters such as surface color and normal vector orientation.

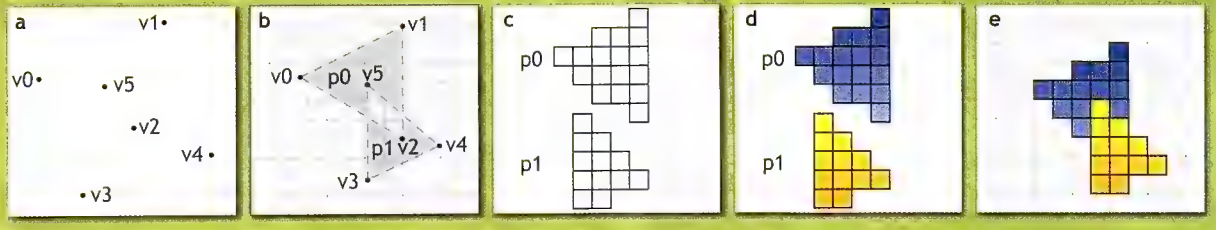
VP (vertex processing). The behavior of VP is application programmable. VP operates on each vertex independently and produces exactly one output vertex record from each input record. One of the most important operations of VP execution is computing the 2D output image (screen) projection of the 3D vertex position.

PG (primitive generation). PG uses vertex topology data provided by the application to group vertices from VP into an ordered stream of primitives (each primitive record is the concatenation of several VP output vertex records). Vertex topology also defines the order of primitives in the output stream.

PP (primitive processing). PP operates independently on each input primitive to produce zero or more output primitives. Thus, the output of PP is a new (potentially longer or shorter) ordered stream of primitives. Like VP, PP operation is application programmable.

FG (fragment generation). FG samples each primitive densely in screen space (this process is called *rasterization*). Each sample is manifest as a fragment record in the FG output stream. Fragment records contain the output image position of the surface sample, its distance from

Graphics Pipeline Operations



(A) Six vertices from the VG output stream define the scene position and orientation of two triangles.

(B) Following VP and PG, the vertices have been transformed into their screen-space positions and grouped into two triangle primitives, *p0* and *p1*.

(C) FG samples the two primitives, producing a set of fragments corresponding to *p0* and *p1*.

(D) FP computes the appearance of the surface at each sample location.

(E) PO updates the output image with contributions from the fragments, accounting for surface visibility. In this example, *p1* is nearer to the camera than *p0*. As a result *p0* is occluded by *p1*.

FIG 2

GPUs
a closer look

the virtual camera, as well as values computed via interpolation of the source primitive's vertex parameters.

FP (fragment processing). FP simulates the interaction of light with scene surfaces to determine surface color and opacity at each fragment's sample point. To give surfaces realistic appearances, FP computations make heavy use of filtered lookups into large, parameterized 1D, 2D, or 3D arrays called *textures*. FP is an application-programmable stage.

PO (pixel operations). PO uses each fragment's screen position to calculate and apply the fragment's contribution to output image pixel values. PO accounts for a sample's distance from the virtual camera and discards fragments that are blocked from view by surfaces closer to the camera. When fragments from multiple primitives contribute to the value of a single pixel, as is often the case when semi-transparent surfaces overlap, many rendering techniques rely on PO to perform pixel updates in the order defined by the primitives' positions in the PP output stream. All graphics APIs guarantee this behavior, and PO is the only stage where the order of entity processing is specified by the pipeline's definition.

SHADER PROGRAMMING

The behavior of application-programmable pipeline stages (VP, PP, FP) is defined by *shader functions* (or shaders). Graphics programmers express vertex, primitive, and fragment shader functions in high-level *shading languages* such as NVIDIA's Cg, OpenGL's GLSL, or Microsoft's HLSL. Shader source is compiled into bytecode offline, then transformed into a GPU-specific binary by the graphics driver at runtime.

Shading languages support complex data types and a rich set of control-flow constructs, but they do *not* contain primitives related to explicit parallel execution. Thus, a shader definition is a C-like function that serially computes output-entity data records from a single input entity. Each function invocation is abstracted as an independent sequence of control that executes in complete isolation from the processing of other stream entities.

As a convenience, in addition to data records from stage input and output streams, shader functions may access (but not modify) large, globally shared data buffers. Prior to pipeline execution, these buffers are initialized to contain shader-specific parameters and textures by the application.

CHARACTERISTICS AND CHALLENGES

Graphics pipeline execution is characterized by the following key properties.

Opportunities for parallel processing. Graphics presents opportunities for both task (across pipeline stages) and data (stages operate independently on stream entities) parallelism, making parallel processing a viable strategy for increasing throughput. Despite abundant potential parallelism, however, constraints on the order of PO stage processing introduce dynamic, fine-grained dependencies that complicate parallel implementation throughout the pipeline. Although output image contributions from *most* fragments can be applied in parallel, those that contribute to the same pixel cannot.

Fixed-function stages encapsulate difficult-to-parallelize work. Each shader function invocation executes serially; programmable stages, however, are trivially parallelizable by executing shader functions simultaneously on multiple stream entities. In contrast, the pipeline's non-programmable stages involve multiple entity interactions (such as ordering dependencies in PO or vertex grouping in PG) and stateful processing. Isolating this non-data-parallel work into fixed stages keeps the shader programming model simple and allows the GPU's programmable processing components to be highly specialized for data-parallel execution. In addition, the separation enables difficult aspects of the graphics computation to be encapsulated in optimized, fixed-function hardware components.

Extreme variations in pipeline load. Although the number of stages and data flows of the graphics pipeline is fixed, the computational and bandwidth requirements of all stages vary significantly depending on the behavior of shader functions and properties of scenes. For example, primitives that cover large regions of the screen generate many more fragments than vertices. In contrast, many small primitives result in high vertex-processing demands. Applications frequently reconfigure the pipeline to use different shader functions that vary from tens of instructions to a few hundred. For these reasons, over

the duration of processing for a single frame, different stages will dominate overall execution, often resulting in bandwidth- and compute-intensive phases of execution. Maintaining an efficient mapping of the graphics pipeline to a GPU's resources in the face of this variability is a significant challenge, as it requires processing and on-chip storage resources to be dynamically reallocated to pipeline stages, depending on current load.

Mixture of predictable and unpredictable data access. The graphics pipeline rigidly defines inter-stage data flows using streams of entities. This predictability presents opportunities for aggregate prefetching of stream data records and highly specialized hardware management on-chip storage resources. In contrast, buffer and texture accesses performed by shaders are fine-grained memory operations on dynamically computed addresses, making prefetch difficult. As both forms of data access are critical to maintaining high throughput, shader programming models explicitly differentiate stream from buffer/texture memory accesses, permitting specialized hardware solutions for both types of accesses.

Opportunities for instruction stream sharing. While the shader programming model permits each shader invocation to follow a unique stream of control, in practice, shader execution on nearby stream elements often results in the same dynamic control-flow decisions. As a result, multiple shader invocations can likely share an instruction stream. Although GPUs must accommodate situations where this is not the case, instruction stream sharing across multiple shader invocations is a key optimization in the design of GPU processing cores and is accounted for in algorithms for pipeline scheduling.

PROGRAMMABLE PROCESSING RESOURCES

A large fraction of a GPU's resources exist within programmable processing cores responsible for executing shader functions. While substantial implementation differences exist across vendors and product lines, all modern GPUs maintain high efficiency through the use of multicore designs that employ both hardware multithreading and SIMD (single instruction, multiple data) processing. As shown in table 1, these throughput-computing techniques are not unique to GPUs (top two rows). In comparison with CPUs, however, GPU designs push these ideas to extreme scales.

Multicore + SIMD Processing = Lots of ALUs. A thread of control is realized by a stream of processor instructions that execute within a processor-managed environment, called an execution (or thread) context. This context consists of states such as a program counter, a stack pointer, general-purpose registers, and virtual memory mappings. A multicore processor replicates processing resources (both ALUs and execution contexts) and organizes them into independent cores. When an application features multiple threads of control, multicore architectures provide increased throughput by executing these instruction streams on each core in parallel. For example, an Intel Core 2 Quad contains four cores and can execute four instruction streams simultaneously. As significant parallelism exists across shader invocations, GPU designs easily push core counts higher. High-end models contain up to 16 cores per chip.

Even higher performance is possible by populating each core with multiple floating-point ALUs. This is done efficiently with SIMD processing, which uses each ALU to perform the same operation on a different piece of data. The most common implementation of SIMD processing is via *explicit* short-vector instructions, similar to those provided by the x86 SSE or PowerPC Altivec ISA extensions. These extensions provide a SIMD width of four, with instructions that control the operation of four ALUs. Alternative implementations, such as NVIDIA's 8-series architecture, perform SIMD execution by *implicitly* shar-

TABLE 1 Tale of the Tape: Throughput Architectures

Type	Processor	Cores/Chip	ALUs/Core[3]	SIMD width	MaxT[4]
GPUs	AMD Radeon HD 2900	4	80	64	48
	NVIDIA GeForce 8800	16	8	32	96
CPUs	Intel Core 2 Quad[1]	4	8	4	1
	STI Cell BE[2]	8	4	4	1
	Sun UltraSPARC T2	8	1	1	4

[1]SSE processing only, does not account for x86 FPU.
[2]Stream processing (SPE) cores only, does not account for PPU cores.
[3]32-bit, floating point (all ALUs are multiply-add except the Intel Core 2 Quad)
[4]The ratio of core thread contexts to simultaneously executable threads. We use the ratio T (rather than the total number of per-core thread contexts) to describe the extent to which processor cores automatically hide thread stalls via hardware multithreading.

GPUs
a closer look

ing an instruction across multiple threads with identical PCs. In either SIMD implementation, the complexity of processing an instruction stream and the cost of circuits and structures to control ALUs are amortized across multiple ALUs. The result is both power- and area-efficient chip execution.

CPU designs have converged on a SIMD width of four as a balance between providing increased throughput and retaining high single-threaded performance. Characteristics of the shading workload make it beneficial for GPUs to employ significantly wider SIMD processing (widths ranging from 32 to 64) and to support a rich set of operations. It is common for GPUs to support SIMD implementations of reciprocal square root, trigonometric functions, and memory gather/scatter operations.

The efficiency of wide SIMD processing allows GPUs to pack many cores densely with ALUs. For example, the NVIDIA GeForce 8800 Ultra GPU contains 128 single-precision ALUs operating at 1.5 GHz. These ALUs are organized into 16 processing cores and yield a peak rate of 384 Gflops (each ALU retires one 32-bit multiply-add per clock). In comparison, a high-end 3-GHz Intel Core 2 CPU contains four cores, each with eight SIMD floating-point ALUs (two 4-width vector instructions per clock), and is capable of, at most, 96 Gflops of peak performance.

GPUs execute groups of shader invocations in parallel to take advantage of SIMD processing. Dynamic per-entity control flow is implemented by executing all control paths taken by the shader invocations. SIMD operations that do not apply to all invocations, such as those within shader code conditional or loop blocks, are partially nullified using write-masks. In this implementation, when shader control flow diverges, fewer SIMD ALUs do useful work. Thus, on a chip with width-*S* SIMD processing, worst-case behavior yields performance equaling *1/S* the chip's peak rate. Fortunately, shader workloads exhibit sufficient levels of instruction stream sharing to justify wide SIMD implementations. Additionally, GPU ISAs contain special instructions that make it possible for shader compilers to transform per-entity control flow into efficient sequences of SIMD operations.

Hardware Multithreading = High ALU Utilization. Thread stalls pose an additional challenge to high-performance shader execution. Threads *stall* (or block) when the processor cannot dispatch the next instruction in an instruction stream because of a dependency on an outstanding instruction. High-latency off-chip memory accesses, most notably those generated by fragment shader texturing operations, cause thread stalls lasting hundreds of cycles (recall that while shader input and output records lend themselves to streaming prefetch, texture accesses do not).

Allowing ALUs to remain idle during the period while a thread is stalled is inefficient. Instead, GPUs maintain more execution contexts on chip than they can simultaneously execute, and they perform instructions from runnable threads when others are stalled. Hardware scheduling logic determines which context(s) to execute in each processor cycle. This technique of overprovisioning cores with thread contexts to hide the latency of thread stalls is called *hardware multithreading*. GPUs use multithreading to hide both memory access and instruction pipeline latencies.

The latency-hiding ability of GPU multithreading is dependent on the ratio of hardware thread contexts to the number of threads that can be simultaneously executed in a clock (value *T* from table 1). Support for more thread contexts allows the GPU to hide longer or more frequent stalls. All modern GPUs maintain large numbers of execution contexts on chip to provide maximal memory latency-hiding ability (*T* ranges from 16 to 96). This represents a significant departure from CPU designs, which attempt to avoid or minimize stalls using large, low-latency data caches and complicated out-of-order execution logic. Current Intel Core 2 and AMD Phenom processors maintain one thread per core, and even high-end models of Sun's multithreaded UltraSPARC T2 processor manage only four times the number of threads they can simultaneously execute.

Note that in the absence of stalls, the throughput of single- and multithreaded processors is equivalent. Multithreading does not increase the number of processing resources on a chip. Rather, it is a strategy that interleaves execution of multiple threads in order to use existing resources more efficiently (improve throughput). On average, a multithreaded core operating at its peak rate runs each thread *1/T* of the time.

Large-scale multithreading requires execution contexts to be compact in order to fit many contexts within on-chip memories. The number of thread contexts supported by a GPU core is shader-program dependent and typically limited by the size of on-chip storage. GPUs require compiled shader binaries to declare input and output entity sizes, as well as bounds on temporary storage and scratch registers required for execution. At runtime, GPUs use these bounds to partition unspillable on-chip storage (including data registers) dynamically among execution contexts. Thus, GPUs support many thread contexts (up to an architecture-specific bound) and, correspondingly, provide maximal latency-hiding ability when shaders use fewer resources. When shaders require large amounts of storage, the number of execution contexts provided by a GPU drops. (The accompanying sidebar details an example of the efficient execution of a fragment shader on a GPU core.)

FIXED-FUNCTION PROCESSING RESOURCES

A GPU's programmable cores interoperate with a collection of specialized fixed-function processing units that provide high-performance, power-efficient implementations of nonshader stages. These components do not simply augment programmable processing; they perform sophisticated operations and constitute an additional hundreds of gigaflops of processing power. Two of the most important operations performed via fixed-function hardware are texture filtering and rasterization (fragment generation).

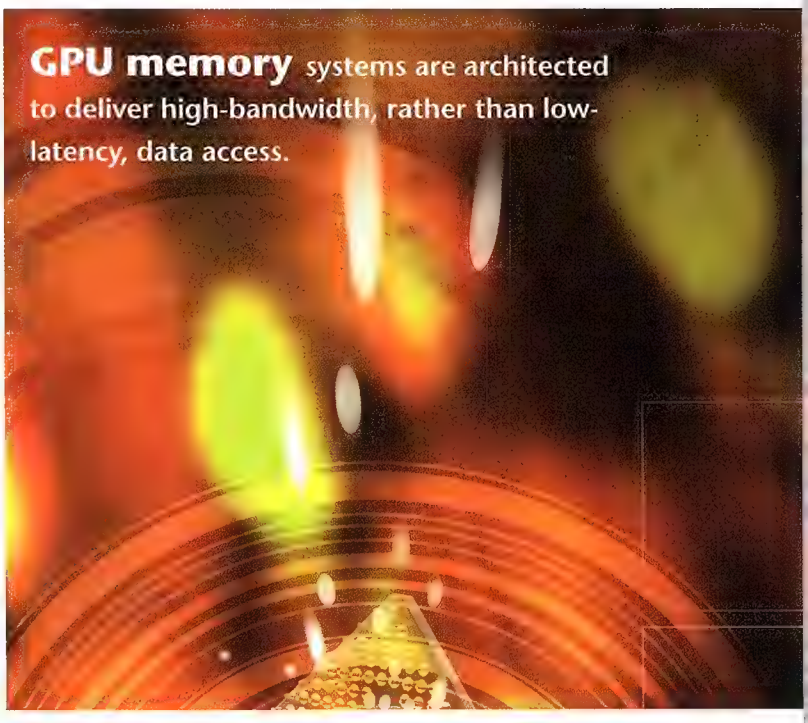
Texturing is handled almost entirely by fixed-function logic. A texturing operation samples a contiguous 1D, 2D, or 3D signal (a texture) that is discretely represented by a multidimensional array of color values (2D texture data is simply an image). A GPU texture-filtering unit accepts a point within the texture's parameterization (represented by a floating-point tuple, such as {.5,.75}) and loads array values surrounding the coordinate from memory. The values are then filtered to yield a single result that represents the texture's value at the specified coordinate. This value is returned to the calling shader function. Sophisticated texture filtering is required for generating high-quality images. As graphics APIs provide a finite set of filtering kernels, and because filtering kernels are computationally expensive, texture filtering is well suited for fixed-function processing.

Primitive rasterization in the FG stage is another key pipeline operation implemented by fixed-function components. Rasterization involves densely sampling a primitive (at least once per output image pixel) to determine which pixels the primitive overlaps. This process involves interpolating the location of the surface at each sample point and then generating fragments for all sample points covered by the primitive. Bounding-box computations and hierarchical techniques optimize the rasterization process. Nonetheless, rasterization involves significant computation.

In addition to the components for texturing and rasterization, GPUs contain dedicated hardware components for operations such as surface visibility determination, output pixel compositing, and data compression/decompression.

THE MEMORY SYSTEM

Parallel-processing resources place extreme load on a GPU's memory system, which services memory requests from both fixed-function and programmable components. These requests include a mixture of fine-granularity and bulk prefetch operations and may even require realtime guarantees (such as display scan out).



Recall that a GPU's programmable cores tolerate large memory latencies via hardware multithreading and that interstage stream data accesses can be prefetched. As a result, GPU memory systems are architected to deliver high-bandwidth, rather than low-latency, data access. High throughput is obtained through the use of wide

GPUs a closer look

memory buses and specialized GDDR (graphics double data rate) memories that operate most efficiently when memory access granularities are large. Thus, GPU memory controllers must buffer, reorder, and then coalesce large numbers of memory requests to synthesize large operations that make efficient use of the memory system. As an example, the ATI HD 2700XT memory controller manipulates thousands of outstanding requests to deliver 105 GB per second of bandwidth from GDDR3 memories attached to a 512-bit bus.

GPU data caches meet different needs from CPU caches. GPUs employ relatively small, read-only caches (no cache coherence) that filter requests destined for the memory controller and reduce bandwidth requirements placed on main memory. Thus, GPU caches typically serve to amplify total bandwidth to processing units rather than decrease latency of memory accesses. Interleaved execution of many threads renders large read-write caches inefficient because of severe cache thrashing. GPUs benefit from small caches that capture spatial locality across simultaneously executed shader invocations. This situation is common, as texture accesses performed while processing fragments in close screen proximity are likely to have overlapping texture-filter support regions.

Although most GPU caches are small, this does not imply that GPUs contain little on-chip storage. Significant amounts of on-chip storage are used to hold entity streams, execution contexts, and thread scratch data.

PIPELINE SCHEDULING AND CONTROL

Mapping the entire graphics pipeline efficiently onto GPU resources is a challenging problem that requires dynamic and adaptive techniques. A unique aspect of GPU computing is that hardware logic assumes a major role in mapping and scheduling computation onto chip resources. GPU hardware "scheduling" logic extends beyond the thread-scheduling responsibilities discussed in previous sections. GPUs automatically assign computations to threads, clean up after threads complete, size and manage buffers that hold stream data, guarantee ordered processing when needed, and identify and discard unnecessary pipeline work. This logic relies heavily on specific upfront knowledge of graphics workload characteristics.

Conventional thread programming uses operating-system or threading API mechanisms for thread creation, completion, and synchronization on shared structures. Large-scale multithreading coupled with the brevity of shader function execution (at most a few hundred instructions), however, means GPU thread management must be performed entirely by hardware logic.

GPUs minimize thread launch costs by preconfiguring execution contexts to run one of the pipeline's three types of shader functions and reusing the configuration multiple times for shaders of the same type. GPUs launch threads when a shader stage's input stream contains a sufficient number of entities, and then they automatically provide threads access to shader input records. Similar hardware logic commits records to the output stream buffer upon thread completion. The distribution of execution contexts to shader stages is reprovisioned periodically as pipeline needs change and stream buffers drain or approach capacity.

GPUs leverage upfront knowledge of pipeline entities to identify and skip unnecessary computation. For example, vertices shared by multiple primitives are identified and VP results cached to avoid duplicate vertex processing. GPUs also discard fragments prior to FP when the fragment will not alter the value of any image pixel. Early fragment discard is triggered when a fragment's sample point is occluded by a previously processed surface located closer to the camera.

Another class of hardware optimizations reorganizes fine-grained operations for more efficient processing. For example, rasterization orders fragment generation to maximize screen proximity of samples. This ordering improves texture cache hit rates, as well as instruction stream sharing across shader invocations. The GPU memory controller also performs automatic reorganization when it reorders memory requests to optimize memory bus and DRAM utilization.

GPUs ensure inter-fragment PO ordering dependencies using hardware logic. Implementations use structures such as post-FP reorder buffers or scoreboards that delay fragment thread launch until the processing of overlapping fragments is complete.

GPU hardware can take responsibility for sophisticated scheduling decisions because semantics and invariants of

Running a Fragment Shader on a GPU Core

Shader compilation to SIMD (single instruction, multiple data) instruction sequences coupled with dynamic hardware thread scheduling leads to efficient execution of a fragment shader on the simplified single-core GPU shown in figure A.

- The core executes an instruction from at most one thread each processor clock, but maintains state for four threads on-chip simultaneously (T=4).
- Core threads issue explicit width-32 SIMD vector instructions; 32 ALUs simultaneously execute a vector instruction in a single clock.
- The core has a pool of 16 general-purpose vector registers (R0 to R15) that are partitioned among thread contexts. The elements of each length-32 vector are 32-bit values.
- The only source of thread stalls is texture access; they have a maximum latency of 50 cycles.

Shader compilation by the graphics driver produces a GPU binary from a high-level fragment shader source. The resulting vector instruction sequence performs 32 invocations of the fragment shader simultaneously by carrying out each invocation in a single lane of the width-32 vectors. The compiled binary requires four vector registers for temporary results and contains 20 arithmetic instructions between each texture access operation.

At runtime, the GPU executes a copy of the shader binary on each of its four thread contexts, as illustrated in figure B. The core executes T0 (thread 0) until it detects a stall resulting from texture access in cycle 20. While T0 waits for the result of the texturing operation, the core continues to execute its remaining three threads. The result of T0's texture access becomes available in cycle 70. Upon T3's stall in cycle 80, the core immediately resumes T0. Thus, at no point during execution are ALUs left idle.

When executing the shader program for this example, a minimum of four threads is needed to keep core ALUs busy. Each thread operates simultaneously on 32 fragments; thus, 4*32=128 fragments are required for the chip to achieve peak performance.

As memory latencies on real GPUs involve hundreds of cycles, modern GPUs must contain support for significantly more threads to sustain high utilization. If we extend our simple GPU to a more realistic size of eight processing cores and provision each core with storage for 16 execution contexts, then simultaneous processing of 4,096 fragments is needed to approach peak processing rates. Clearly, GPU performance relies heavily on the abundance of parallel shading work.

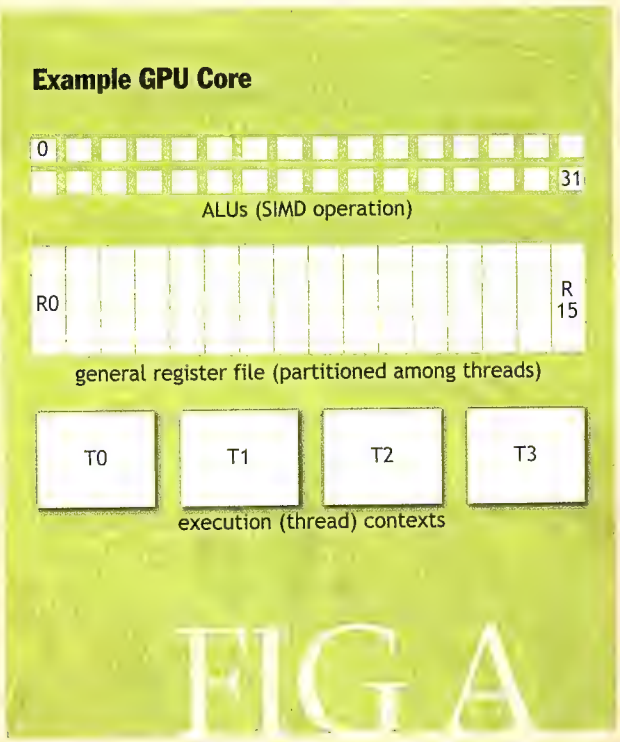


FIG A

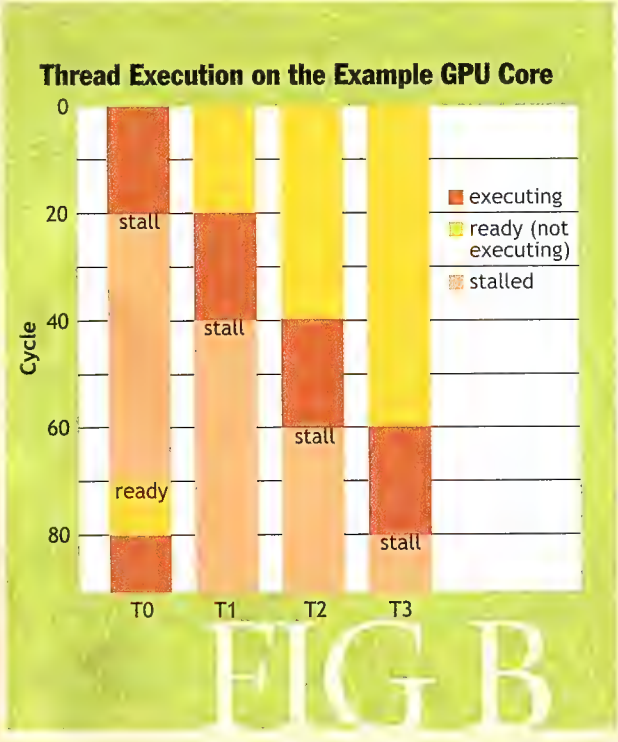


FIG B

GPUs
a closer look

the graphics pipeline are known a priori. Hardware implementation enables fine-granularity logic that is informed by precise knowledge of both the graphics pipeline and the underlying GPU implementation. As a result, GPUs are highly efficient at using all available resources. The drawback of this approach is that GPUs execute only those computations for which these invariants and structures are known.

Graphics programming is becoming increasingly versatile. Developers constantly seek to incorporate more sophisticated algorithms and leverage more configurable graphics pipelines. Simultaneously, the growing popularity of GPGPU (general-purpose computing using GPU platforms) has led to new interfaces for accessing GPU resources. Given both of these trends, the extent to which GPU designers can embed a priori knowledge of computations into hardware scheduling logic will inevitably decrease over time.

A major challenge in the evolution of GPU programming involves preserving GPU performance levels while increasing the generality and expressiveness of application interfaces. The designs of GPGPU interfaces, such as NVIDIA's CUDA and AMD's CAL, are evidence of how difficult this challenge is. These frameworks abstract computation as large batch operations that involve many invocations of a kernel function operating in parallel. The resulting computations execute on GPUs efficiently only under conditions of massive data parallelism. Programs that attempt to implement non-data-parallel algorithms perform poorly.

GPGPU programming models are simple to use and permit well-written programs to make good use of both GPU programmable cores and (if needed) texturing resources. Programs using these interfaces, however, cannot use powerful fixed-function components of the chip, such as those related to compression, image compositing, or rasterization. Also, when these interfaces are enabled, much of the logic specific to graphics-pipeline scheduling is simply turned off. Thus, current GPGPU programming frameworks restrict computations so that their structure, as well as their use of chip resources, remains sufficiently simple for GPUs to run these programs in parallel.

GPU AND CPU CONVERGENCE

The modern graphics processor is a powerful computing platform that resides at the extreme end of the design space of throughput-oriented architectures. A GPU's processing resources and accompanying memory system are heavily optimized to execute large numbers of operations in parallel. In addition, specialization to the graphics domain has enabled the use of fixed-function processing and allowed hardware scheduling of a parallel computation to be practical. With this design, GPUs deliver unsurpassed levels of performance to challenging workloads while maintaining a simple and convenient programming interface for developers.

Today, commodity CPU designs are adopting features common in GPU computing, such as increased core counts and hardware multithreading. At the same time, each generation of GPU evolution adds flexibility to previous high-throughput GPU designs. Given these trends, software developers in many fields are likely to take interest in the extent to which CPU and GPU architectures and, correspondingly, CPU and GPU programming systems, ultimately converge. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

KAYVON FATAHALIAN is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming systems for commodity parallel architectures and computer graphics/animation systems for the interactive and film domains. His thesis research seeks to enable execution of more flexible rendering pipelines on future GPUs and multicore PCs. He will soon be looking for a job.

MIKE HOUSTON is a Ph.D. candidate in computer science in the Computer Graphics Laboratory at Stanford University. His research interests include programming models, algorithms, and runtime systems for parallel architectures including GPUs, Cell, multicore CPUs, and clusters. His dissertation includes the Sequoia runtime system, a system for programming hierarchical memory machines. He received his B.S. in computer science from UCSD in 2001 and is a recipient of the Intel Graduate Fellowship.

BETTER SOFTWARE
CONFERENCE & EXPO

JUNE 9-12, 2008
LAS VEGAS, NEVADA
The Venetian

KEYNOTES BY INTERNATIONAL EXPERTS

Jean Tabaka
Rally Software Development

Johanna Rothman
Rothman Consulting Group, Inc.

Michael Mah
QSM Associates

Ken Schwaber
Advanced Development Methods, Inc.

TUTORIALS WORKSHOPS CLASSES

CELEBRATING 5 FIVE YEARS

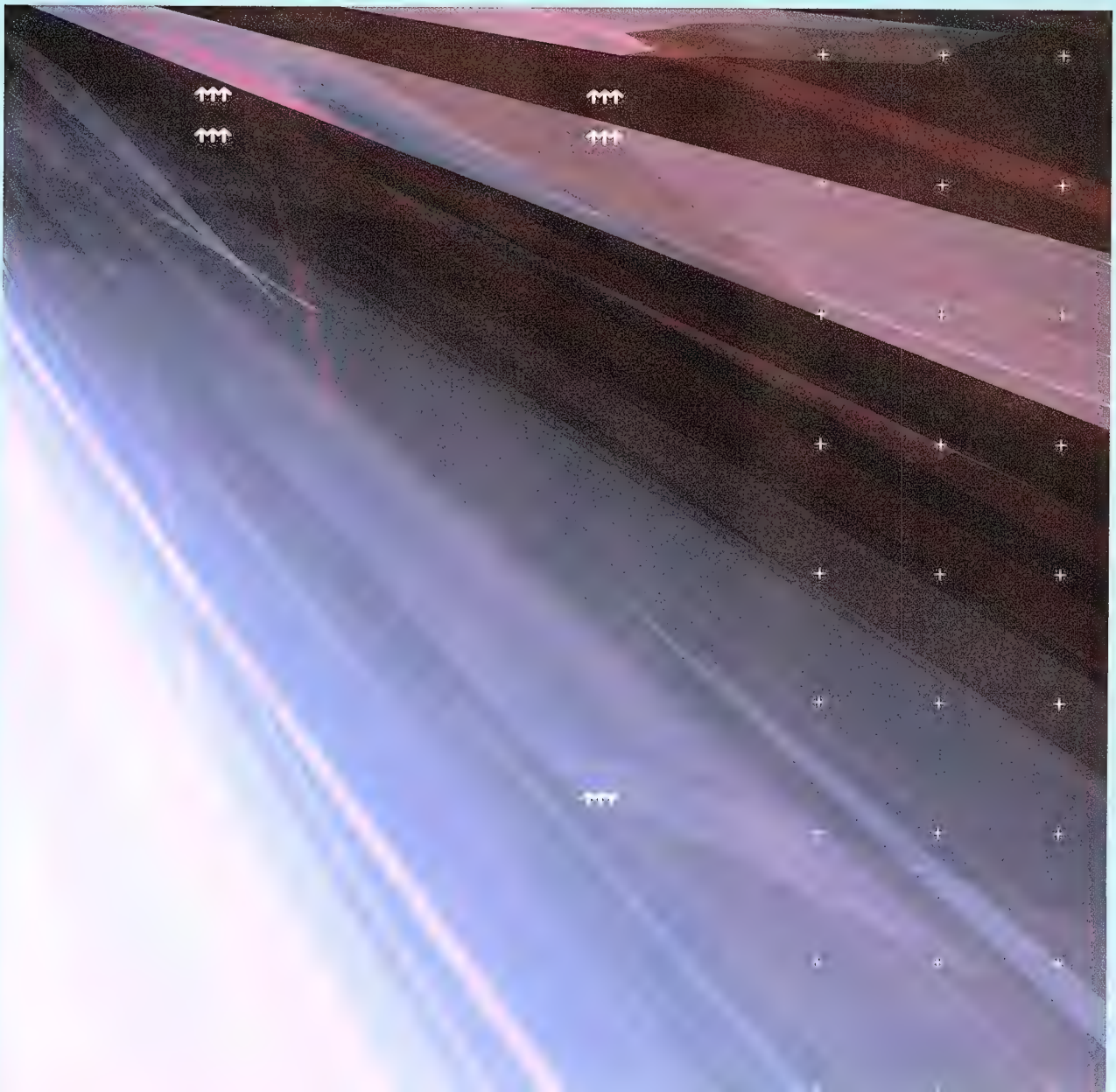
Alan Shalloway
Andy Glover
Andy Hunt
Andy Kaufman
Beth Layman
Bob Hartman
Chuck Allison
Dan North
David Herron
David Spann
Ed Weller
Hubert Smits
James McCaffrey
Jean Tabaka
Jeff Patton
Jeff Payne
Johanna Rothman
Julie Gardiner
Ken Pugh
Ken Schwaber
Kent McDonald
Lee Copeland
Lee Devin
Linda Rising
Linda Westfall
Lisa Crispin
Michael Mah
Michael Sutton
Michele Sliger
Paco Hope
Pollyanna Pixton
Rebecca Wirfs-Brock
Richard Bender
Rob Myers
Robert Galen
Stacia Broderick
Timothy Korson
Todd Little
Will McKnight

SOFTWARE QUALITY ENGINEERING
www.sqe.com

www.sqe.com/bsce

Data-Parallel Computing

CHAS. BOYD, MICROSOFT



Data parallelism is a key concept in leveraging the power of today's manycore GPUs.

Users always care about performance. Although often it's just a matter of making sure the software is doing only what it should, there are many cases where it is vital to get down to the metal and leverage the fundamental characteristics of the processor.

Until recently, performance improvement was not difficult. Processors just kept getting faster. Waiting a year for the customer's hardware to be upgraded was a valid optimization strategy. Nowadays, however, individual processors don't get much faster; systems just get more of them.

Much comment has been made on coding paradigms to target multiple-processor cores, but the data-parallel paradigm is a newer approach that may just turn out to be easier to code to, and easier for processor manufacturers to implement.

This article provides a high-level description of data-parallel computing and some practical information on how and where to use it. It also covers data-parallel programming environments, paying particular attention to those based on programmable graphics processors.

A BIT OF BACKGROUND

Although the rate of processor-performance growth seems almost magical, it is gated by fundamental laws of physics. For the entire decade of the '90s, these laws enabled processors to grow exponentially in performance as a result of improvements in gates-per-die, clock speed, and instruction-level parallelism. Beginning in 2003, though, the laws of physics (power and heat) put an end to growth in clock speed. Then the silicon area requirements

Data-Parallel Computing

for increasingly sophisticated ILP (instruction-level parallelism) schemes (branch prediction, speculative execution, etc.) became prohibitive. Today the only remaining basis for performance improvement is gate count.

Recognizing this, manufacturers have restructured to stop pushing clock rate and focus on gate count. Forecasts project that gates-per-die can double every two years for the next six to eight years at least. What do you do with all those gates? You make more cores. The number of cores per die will therefore double every two years, resulting in four times today's core counts (up to 32 cores) by 2012.

Customers will appreciate that growth rate, but they will benefit only if software becomes capable of scaling across all those new cores. This is the challenge that performance software faces in the next five to ten years. For the next decade, the limiting factor in software performance will be the ability of software developers to restructure code to scale at a rate that keeps up with the rate of core-count growth.

PARALLEL PROGRAMMING

Parallel programming is difficult. We deprecate the use of GOTO statements in most languages, but parallel execution is like having them randomly sprinkled throughout the code during execution. The assumptions about order of execution that programmers have made since their early education no longer apply.

The single-threaded von Neumann model is comprehensible because it is deterministic. Parallel code is subject to errors such as deadlock and livelock, race conditions, etc. that can be extremely subtle and difficult to identify, often because the bug is nonrepeatable. These issues are so severe that despite decades of effort and dozens of different approaches, none has really gained significant adoption or even agreement that it is the best solution to the problem.

An equally subtle challenge is performance scaling. Amdahl's law states that the maximum speedup attainable by parallelism is the reciprocal of the proportion of code that is not parallelizable. If 10 percent of a given code base is not parallel, even on an infinite number of processors it cannot attain more than a tenfold speedup.

Although this is a useful guideline, determining how much of the code ends up running in parallel fashion is very difficult. Serialization can arise unexpectedly as a result of contention for a shared resource or requirements to access too many distant memory locations.

The traditional methods of parallel programming (thread control via locks, message-passing interface, etc.) often have limited scaling ability because these mechanisms can require serialization phases that actually increase with core count. If each core has to synchronize with a single core, that produces a linear growth in serial code, but if each core has to synchronize with all other cores, there can be a combinatoric increase in serialization.

After all, any code that serializes is four times slower on a four-core machine, but 40 times slower on a 40-core machine.

Another issue with performance scaling is more fundamental. A common approach in multicore parallel programming for games is to start with a top-down breakdown. Relatively isolated subsystems are assigned to separate cores, but what happens once the number of subsystems in the code base is reached? Since restructuring code at this level can be pervasive, it often requires a major rewrite to break out subsystems at the next finer level, and again for each hardware generation.

For all these reasons, transitioning a major code base to parallel paradigms is time consuming. Getting all the subtle effects of nondeterminism down to an acceptable level can take years. It is likely that by that time, core-count growth will have already exceeded the level of parallelism that the new code structure can scale to. Unfortunately, the rate of core-count growth may be outstripping our ability to adapt to it.

Thus, the time has come to look for a new paradigm—ideally one that scales with core count but without requiring restructuring of the application architecture every time a new core count is targeted. After all, it's not about choosing a paradigm that operates well at a fixed core count; it's about choosing one that continues to scale with an increasing number of cores without requiring code changes. We need to identify a finer level of granularity for parallelism.

DATA-PARALLEL PROGRAMMING

Given the difficulty of finding enough subsystem tasks to assign to dozens of cores—the only elements of which there are a comparable number are data elements—the data-parallel approach is simply to assign an individual data element to a separate logical core for processing. Instead of breaking code down by subsystems, we look for fine-grained inner loops within each subsystem and parallelize those.

For some tasks, there may be thousands to millions of data elements, enabling assignment to thousands of cores. (Although this may turn out to be a limitation in the future, it should enable code to scale for another decade or so.) For example, a modern GPU can support hundreds of ALUs (arithmetic logic units) with hundreds of threads per ALU for nearly 10,000 data elements on the die at once.

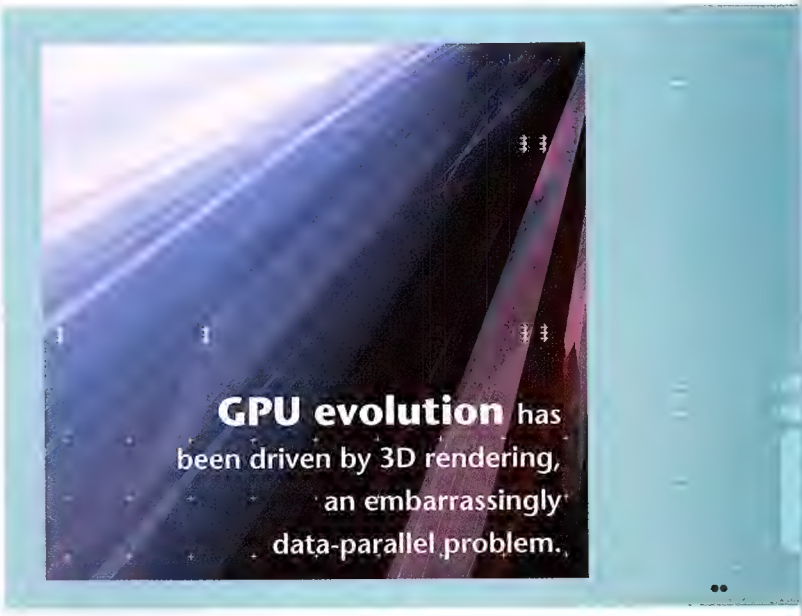
The history of data-parallel processors began with the efforts to create wider and wider vector machines. Much of the early work on both hardware and data-parallel algorithms was pioneered at companies such as MasPar, Tera, and Cray.

Today, a variety of fine-grained or data-parallel programming environments are available. Many of these have achieved recent visibility by supporting GPUs. They can be categorized as follows:

Older languages (C*, MPL, Co-Array Fortran, Cilk, etc.). Several languages have been developed for fine-grained parallel programming and vector processing. Many add only a very small difference in syntax from well-known languages. Few of them support a variety of platforms and they may not be available commercially or be supported long term as far as updates, documentation, and materials.

Newer languages (XMT-C, CUDA, CAL, etc.). These languages are being developed by the hardware company involved and therefore are well supported. They are also very close to current C++ programming models syntactically; however, this can cause problems because the language then provides no explicit representation of the unique aspects of data-parallel programming or the processor hardware. Although this can reduce the changes required for an initial port, the resulting code hides the parallel behavior, making it harder to comprehend, debug, and optimize. Simplifying the initial port of serial code through syntax is not that useful to begin with, since for best performance it is often an entire algorithm that must be replaced with a data-parallel version. Further, in the interest of simplicity, these APIs may not expose the full features of the graphics-specific silicon, which implies an underutilized silicon area.

Array-based languages (RapidMind, Acceleware, Microsoft Accelerator, Ct, etc.). These languages are based on array data types and specific intrinsics that operate on them. Algorithms converted to these languages often result in code that is shorter, clearer, and very likely faster than before. The challenge of restructuring design concepts into array paradigms, however, remains a barrier to adoption of these languages because of the high level of abstraction at which it must be done.



Graphics APIs (OpenGL, Direct3D). Recent research in GPGPU (general-purpose computing on graphics processing units) has found that while the initial ramp-up of using graphics APIs can be difficult, they do provide a direct mapping to hardware that enables very specific optimizations, as well as access to hardware features that other approaches may not allow. For example, work by Naga Govindaraju[1] and Jens Krüger[2] relies on access to fixed-function triangle interpolators and blending units that the newer languages mentioned here often do not expose. Further, there is good commercial support and a large and experienced community of developers already using them.

GPUS AS DATA-PARALLEL MACHINES

The GPU is the second-most-heavily used processor in a typical PC. It has evolved rapidly over the past decade to reach performance levels that can exceed the CPU by

Data-Parallel Computing

a large factor, at least on appropriate workloads.[3] GPU evolution has been driven by 3D rendering, an embarrassingly data-parallel problem, which makes the GPU an excellent target for data-parallel code. As a result of this significantly different workload design point (processing model, I/O patterns, and locality of reference), the GPU has a substantially different processor architecture and memory subsystem design, typically featuring a broader SIMD (single instruction, multiple data) width and a higher-latency, higher-bandwidth streaming memory system. The processing model exposed via a graphics API is a task-serial pipeline made up of a few data-parallel stages that use no interthread communication mechanisms at all. While separate stages appear for processing vertices or pixels, the actual architecture is somewhat simpler.

As shown in figure 1, a modern DirectX10-class GPU has a single array of processors that perform the computational work of each stage in conjunction with specialized hardware. After polygon-vertex processing, a specialized hardware interpolator unit is used to turn each polygon into pixels for the pixel-processing stage. This unit can be thought of as an address generator. At the end of the pipeline, another specialized unit blends completed pixels into the image buffer. This hardware is often useful in accumulating results into a destination array. Further, all processing stages have access to a dedicated texture-sampling unit that performs linearly interpolated reads on 1D, 2D, or 3D source arrays in a variety of data-element formats.

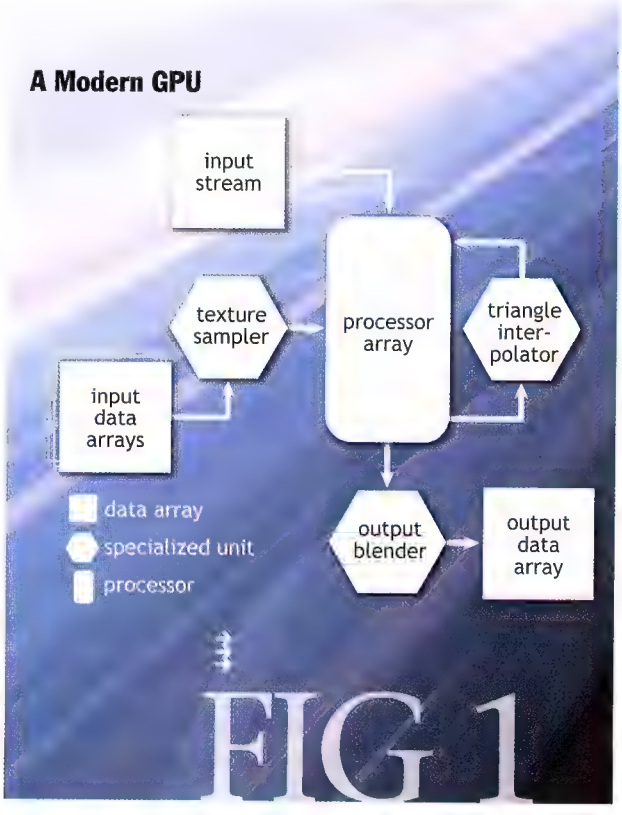


FIG 1

Shaped by these special workload requirements, the modern GPU has:

- Ten times the GFLOPS of CPU chips for similar price and power consumption
- Thousands of threads distributed over hundreds of single-precision floating-point ALUs
- A dedicated streaming-memory system with 10 times the memory bandwidth of a CPU
- Dedicated memory capacity similar to the CPU system memory capacity
- Specialized cores for filtering, blending, rasterizing, and video processing

A GPU's memory subsystem is designed for higher I/O latency to achieve increased throughput. It assumes only very limited data reuse (locality in read/write access), featuring small input and output caches designed more as FIFO (first in, first out) buffers than as mechanisms to avoid round-trips to memory.

Recent research has looked into applying these processors to other algorithms beyond 3D rendering. There have been applications that have shown significant benefits over CPU code. In general, those that most closely match the original design workload of 3D graphics (such as image processing) and can find a way to leverage either the tenfold compute advantage or the tenfold bandwidth advantage have done well. (Much of this work is cataloged on the Web at http://www.gpgpu.org.)

This research has identified interesting algorithms. For example, compacting an array of variable-length records is a task that has a data-parallel implementation on the parallel prefix sum or scan. The prefix-sum algorithm computes the sum of all previous array elements (i.e.,

the first output element in a row r is r_0, while the second is $o_1 = r_0 + r_1$, and the nth output element is $o_n = r_0 + r_1 + \ldots + r_n$). Using this, a list of record sizes can be accumulated to compute the absolute addresses where each record element is to be written. Then the writes can occur completely in parallel. Note that if the writes are done in order, the memory-access pattern is still completely sequential.[4]

MAKING CODE DATA-PARALLEL

Before starting to write your code, check for tasks that are known data-parallel cases. Often you can find library routines already available for accelerating common tasks using data-parallel hardware. Most data-parallel programming environments include such libraries as a convenient way for users to begin adopting their technology.

If you need to write custom data-parallel code, the process is similar to a localized optimization effort. You can adopt data-parallel programming incrementally, since you can identify and optimize the key inner loops one at a time, without perturbing the larger-scale structure of the code base. Here are the basic steps for converting code to the data-parallel model:

1. Identify a key task that looks data-parallel.
2. Identify a data-parallel algorithm for this task.
3. Select a data-parallel programming environment.
4. Implement code.
5. Evaluate performance scaling rate.
6. Go to step 1.

STEP 1: IDENTIFY A KEY TASK THAT LOOKS DATA-PARALLEL

Look for a segment of code that doesn't rely greatly on cross communication between data elements, or conversely, a set of data elements that can be processed without requiring too much knowledge of each other. Look for data-access patterns that can be regularized, as opposed to arbitrary/random (such as linear arrays versus sparse-tree data structures).

While searching for candidates to parallelize, you can evaluate performance potential via Amdahl's law: just comment out this candidate task (simulate infinite parallelism) and check to see total performance change. If there isn't a significant improvement, going through the effort of parallelizing won't pay off.

STEP 2: IDENTIFY A DATA-PARALLEL ALGORITHM FOR THIS TASK

Often a good place to look is in the history books (math) or in routines developed by Tera/Cray for its vector processors. For example, bitonic sorts were identified as interesting before computers were developed, but fell out of favor during the rise of current cache-based machines. Other examples are radix sorts, and prefix sum (scan) operations used for packing sparse data.

STEP 3: SELECT A DATA-PARALLEL PROGRAMMING ENVIRONMENT

Many data-parallel programming environments are available today. Many of the criteria to use in evaluation are the same as for any development environment. The areas to look for are:

- **Abstraction level.** Do you need a library, a set of data-abstraction utilities, or a language?
- **Syntax clarity.** Are limitations of the implementation explicit in the syntax or hidden by it?
- **Maintainability.** Would the resulting code complexity be manageable?
- **Support.** Are there user groups or support services?
- **Availability.** How broadly distributed is the environment or any hardware that it requires?
- **Compatibility.** Is the environment compatible with a broad range of systems or only a specific subset?
- **Lifespan.** Is the environment compatible with future hardware, even from the same vendor?
- **Documentation.** Do the docs make sense? Are the samples useful?
- **Cost.** How much will it cost users of your product to get any required hardware or software?

STEP 4: IMPLEMENT CODE

Code it up, at least at the pseudocode level. If implementation turns out to require more than one or two places where interthread communication is required, then this may not be a sufficiently data-parallel algorithm. In that case, it may be necessary to look for another algorithm (step 2) or another task to parallelize (step 1).

STEP 5: EVALUATE PERFORMANCE SCALING

Performance at a given core count is interesting but not the key point. (If you are going to check that, be sure to compare using a realistic "before" case.) A more important metric to check is how the new code scales with increasing core count. If there is no sign of a performance plateau, the system will have some scaling headroom. After all, absolute performance relative to a single core is not as relevant as how it scales with core-count growth over time.

Data-Parallel Computing

In summary:

- **Understand the paradigm.** What are data-parallel computation and the streaming-memory model?
- **Understand your code.** Which portions operate at which level of granularity?
- **Understand the environment.** How does it help solve the problem?

GPU PERFORMANCE HINTS

If targeting a GPU, are there operations that can leverage the existing graphics-related hardware? Are your data types small enough? GPUs are designed to operate on small data elements so media data (image/video pixels or audio samples) is a good fit. Or when sorting on the GPU, working with key-index pairs separately is often a win. Then the actual movement of data records can be done on the CPU, or on the GPU as a separate pass.

GPUs are optimized for work with 1D, 2D, or 3D arrays of similar data elements. Array operations are often faster using GPU hardware because it can transparently optimize them for spatially coherent access.

When reading such arrays, the GPU can easily linearly interpolate regular array data. This effectively enables a floating-point (fuzzy) array index. Many mathematical algorithms use either a simple linear interpolation of array elements or slightly higher-order schemes that can be implemented as a few linear interpolations. GPU hardware has a significant proportion of silicon allocated to optimizing the performance of these operations.

Algorithms that involve an accumulation or summation of values into a set of results (instead of just a write/copy) can leverage yet another large chunk of special silicon on GPUs: the blender is designed for efficiently compositing or accumulating values into an array. Some matrix math algorithms and reduction operations have shown benefits here.

REGISTER PRESSURE

Some architectures (such as GPUs) are flexible in that they can assign variable numbers of threads to a core based on how many registers each thread uses. This enables more threads to be used when fewer temporary registers are needed, but reduces the threads available (and the parallelism) for algorithms that need more registers. The key is to break tasks into simpler steps that can be executed across even more parallel threads. This is the essence of data-parallel programming.

For example, a standard 8x8 image DCT (discrete cosine transform) algorithm operates on transposed data for its second half. The transpose can takes dozens of registers to execute in place, but breaking it into two passes so that the transpose happens in the intervening I/O results in only a handful of registers needed for each half. This approach improved performance from far slower than a CPU to three times that of a highly optimized SSE assembly routine.

HINTS FOR REDUCTIONS

Reductions are common operations: find the total, average, min, max, or histogram of a set of data. The computations are easily data-parallel, but the output write is an example of cross-thread communication that must be managed carefully

Initial implementations allocated a single shared location for all the threads to write into, but execution was completely serialized by write contention to that location. Allocating multiple copies of the reduction destination and then reducing these down in a separate step was found to be much faster. The key is to allocate enough intermediate locations to cover the number of cores (hundreds) and, therefore, performance level that you want to scale to.

PROGRAMMING THE MEMORY SUBSYSTEM

The data-parallel paradigm extends to the memory subsystem as well. A full data-parallel machine is able not only to process individual data elements separately, but also to read and write those elements in parallel. This characteristic of the memory subsystem is as important to performance as the execution model. For example, I/O ports are a shared resource, and performance is improved if multiple threads are not contending for the same one.

Data structures manipulated imply memory-access patterns. We have seen cases where switching from pointer-based data structures such as linked lists or sparse trees to data-parallel-friendly ones (regular arrays, grids, packed

streams, etc.) allows code to become compute-bound instead of memory-bound (which can be as much as 10 times faster on GPUs). This is because memory is typically organized into pages, and there is some overhead in switching between pages. Grouping data elements and threads so that many results can be read from (or written to) the same page helps with performance.

Many types of trees and other sparse-data structures have data-parallel-friendly array-based implementations. Although using these structures is quite conventional, their implementations are nonintuitive to developers trained on pointer-based schemes.[5]

The most important characteristic of the GPU memory subsystem is the cache architecture. Unlike a CPU, the GPU has hardly any read/write cache. It is assumed that so much data will be streaming through the processor that it will overflow just about any cache. As a result, the only caches present are separate read-through and write-through buffers that smooth out the data flow. Therefore, it is critical to select algorithms that do not rely on reuse of data at scales larger than the few local registers available. For example, histogram computation requires more read/write storage to contain the histogram bins than typical register allocation supports. Upcoming GPU architectures are beginning to add read/write caches so that more algorithms will work, including reasonably sized histograms, but since these caches are still 10 to 100 times smaller than those on the CPU, this will remain a key criterion when choosing an algorithm.

GPUS AS DATA-PARALLEL HARDWARE

GPU systems are cheap and widely available, and many programmers (such as game developers) have identified key approaches to programming them efficiently.

First, it can be important to leverage all the silicon on the die. Applications that don't light up the graphics-specific gates are already at a disadvantage compared with a CPU. For example, Govindaraju's sort implementations show significant benefits from using the blending hardware.[6]

Another way to ensure programming efficiency is to keep the data elements small. This extra hardware is assuming graphics data types that are optimal when they are 16 or fewer bytes in size, and ideally four bytes. If you can make your data look like what a GPU usually processes, you will get large benefits.

Unfortunately, the GPU's high-speed memory system (10 times faster throughput than the CPU front side bus) is typically connected to the CPU by a link that is 10 times slower than CPU memory. Minimizing data and control traffic through this link is vital to GPU performance in low-latency scenarios. The secret is to keep data in the GPU's memory as long as possible, bringing it back to the CPU only for persistent storage. Sometimes this may involve executing a small non-data-parallel task on the GPU because the cost of sending the required data across to the CPU, synchronizing it, and sending it back may be even greater.

GPU GENERALITY

With shorter design cycles, GPUs have been evolving more rapidly than CPUs. This evolution has typically been in the direction of increased generality. Now we are seeing GPU generality growing beyond the needs of basic rendering to more general applications. For example, in the past year new GPU environments have become available that expose features that the graphics APIs do not. Some now support sharing of data among threads and more flexible memory-access options.

This enables entirely new classes of algorithms on GPUs. Most obviously, more general approaches to 3D processing are becoming feasible, including manipulation of acceleration data structures for ray tracing, radiosity, or collision detection. Other obvious applications are in media processing (photo, video, and audio data) where the data types are similar to those of 3D rendering. Other

Fundamental Algorithms Sorted by Granularity of Parallelism

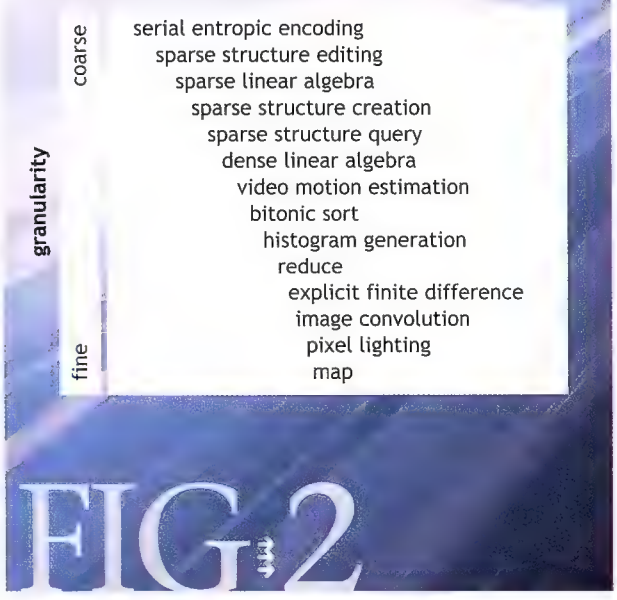


FIG. 2

Data-Parallel Computing

domains using similar data types are seismic and medical analysis.

FUTURE HARDWARE EVOLUTION: CPU/GPU CONVERGENCE?

Processor features such as instruction formats will likely converge as a result of pressure for a consistent programming model. GPUs may migrate to narrower SIMD widths to increase performance on branching code, while CPUs move to broader SIMD width to improve instruction efficiency.

The fact remains, however, that some tasks can be executed more efficiently using data-parallel algorithms. Since efficiency is so critical in this era of constrained power consumption, a two-point design that enables the optimal mapping of tasks to each processor model may persist for some time to come.

Further, if the hardware continues to lead the software, it is likely that systems will have more cores than the application can deal with at a given point in time, so providing a choice of processor types increases the chance of more of them being used.

Conceivably, a data-parallel system could support the entire feature set of a modern serial CPU core, including a rich set of interthread communications and synchronization mechanisms. The presence of such features, however, may not matter in the longer term because the more such traditional synchronization features are used, the worse performance will scale to high core counts. The fastest apps are not those that port their existing single-threaded or even dual-threaded code across, but those that switch to a different parallel algorithm that scales better because it relies less on general synchronization capabilities.

Figure 2 shows a list of algorithms that have been implemented using data-parallel paradigms with varying degrees of success. They are sorted roughly in order of how well they match the data-parallel model.

Data-parallel processors are becoming more broadly available, especially now that consumer GPUs support data-parallel programming environments. This paradigm shift presents a new opportunity for programmers who adapt in time.

The data-parallel industry is evolving without much guidance from software developers. The first to arrive will have the best chance to drive and shape upcoming data-parallel hardware architectures and development environments to meet the needs of their particular application space.

When programmed effectively, GPUs can be faster than current PC CPUs. The time has come to take advantage of this new processor type by making sure each task in your code base is assigned to the processor and memory model that is optimal for that task. Q

REFERENCES

1. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D. 2006. GPUTeraSort: High-performance graphics coprocessor sorting for large database management. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data;* http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2005-183).
2. Krüger, J., Westermann, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics* 22(3).
3. Blythe, D. 2008. The Rise of the GPU. *Proceedings of the IEEE* 96(5).
4. Shubhabrata, S., Lefohn, A.E., Owens, J.D. 2006. A work-efficient step-efficient prefix sum algorithm. *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*: D-26-27.
5. Lefohn, A.E., Kniss, J., Strzodka, R., Sengupta, S., Owens, J.D. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics 25(1).*
6. See reference 1.

SUGGESTED FURTHER READING

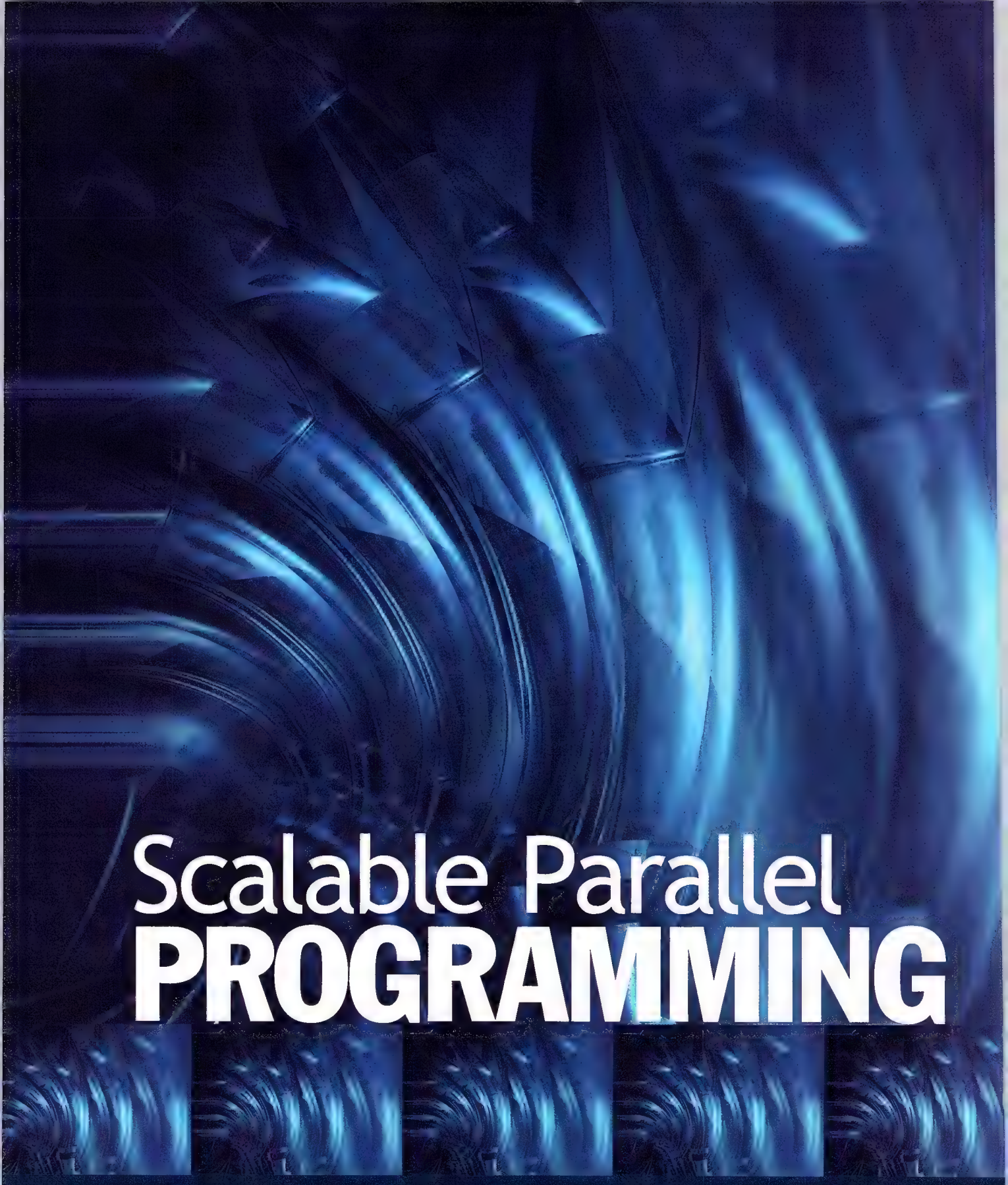
GPU Gems 2:
http://developer.nvidia.com/object/gpu_gems_2_home.html
GPU Gems 3:
http://developer.nvidia.com/object/gpu-gems-3.html Ch 39 on prefix sum
Glift data structures:
http://graphics.cs.ucdavis.edu/~lefohn/work/glift/
Rapidmind:

http://www.rapidmind.net/index.php
Intel Ct:
http://www.intel.com/research/platform/terascale/TeraScale_whitepaper.pdf
Microsoft DirectX SDK:
http://msdn2.microsoft.com/en-us/library/aa139763.aspx
Direct3D HLSL:
http://msdn2.microsoft.com/en-us/library/bb509561.aspx
Nvidia CUDA SDK:
http://developer.nvidia.com/object/cuda.html
AMD Firestream SDK:
http://ati.amd.com/technology/streamcomputing/stream-computing.pdf
Microsoft Research's Accelerator:
http://research.microsoft.com/research/pubs/view.aspx?type=technical%20report&id=1040&0sr=a
http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx

LOVE IT, HATE IT? LET US KNOW

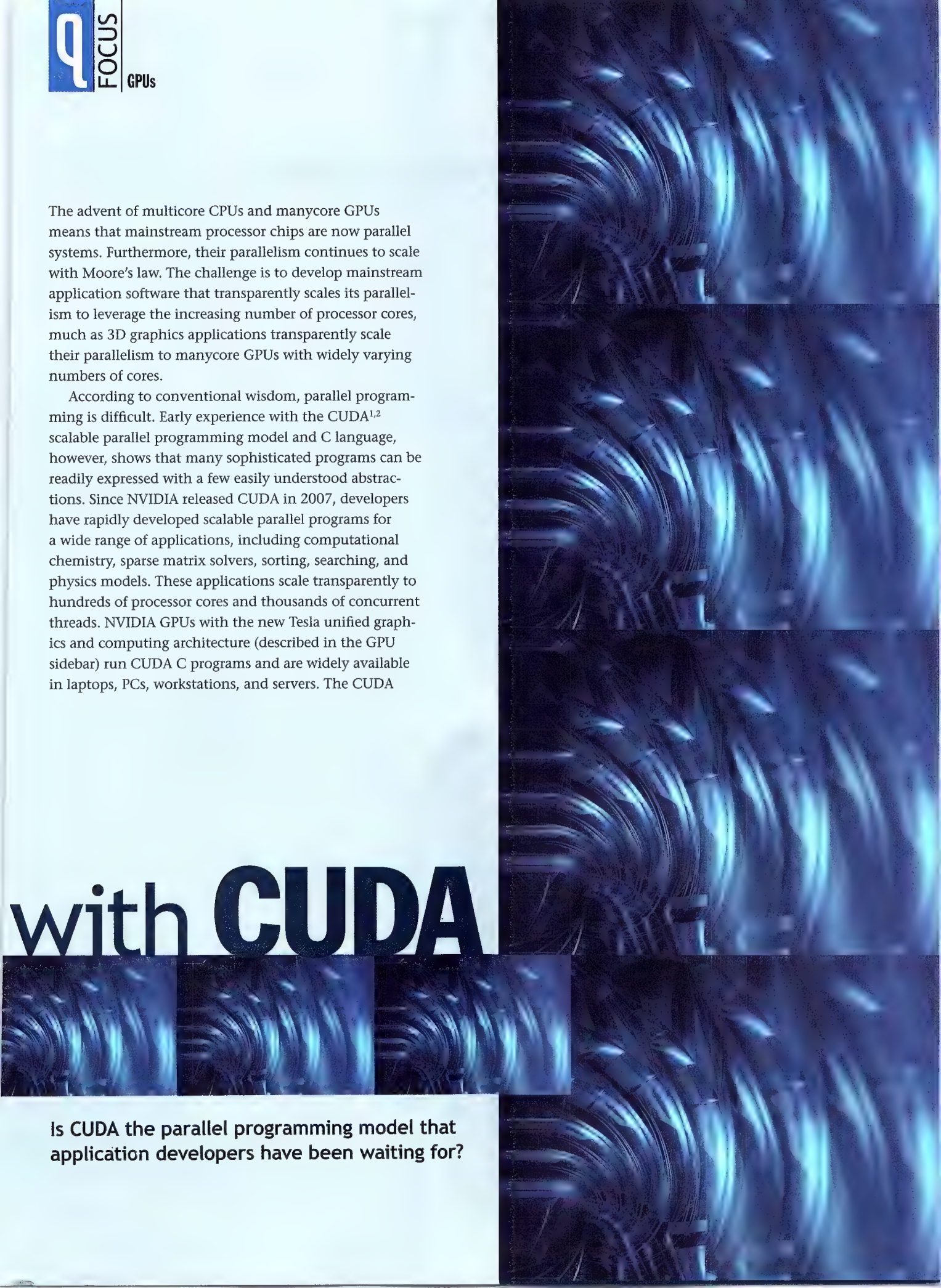
feedback@acmqueue.com or www.acmqueue.com/forums

CHAS. BOYD is a software architect at Microsoft. He joined the Direct3D team in 1995 and has contributed to releases since DirectX 3. During that time he has worked closely with hardware and software developers to drive the adoption of features such as programmable hardware shaders and float pixel processing into consumer graphics. Recently he has been investigating new processing architectures and applications for mass-market consumer systems.



Scalable Parallel PROGRAMMING

JOHN NICKOLLS, IAN BUCK, AND
MICHAEL GARLAND, NVIDIA,
KEVIN SKADRON, UNIVERSITY OF VIRGINIA



with CUDA

Is CUDA the parallel programming model that application developers have been waiting for?

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop mainstream application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

According to conventional wisdom, parallel programming is difficult. Early experience with the CUDA[1,2] scalable parallel programming model and C language, however, shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads. NVIDIA GPUs with the new Tesla unified graphics and computing architecture (described in the GPU sidebar) run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA

Scalable Parallel **PROGRAMMING** with **CUDA**

model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs.[3]

CUDA provides three key abstractions—a hierarchy of thread groups, shared memories, and barrier synchronization—that provide a clear parallel structure to conventional C code for one thread of the hierarchy. Multiple levels of threads, memory, and synchronization provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into

UNIFIED GRAPHICS AND COMPUTING GPUS

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU (graphics processing unit) has evolved into a highly parallel, multithreaded, manycore processor. It is designed to efficiently support the graphics shader programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels at fine-grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently.

The tremendous raw performance of modern GPUs has led researchers to explore mapping more general non-graphics computations onto them. These GPGPU (general-purpose computation on GPUs) systems have produced some impressive results, but the limitations and difficulties of doing this via graphics APIs are legend. This desire to use the GPU as a more general parallel computing device motivated NVIDIA to develop a new unified graphics and computing GPU architecture and the CUDA programming model.

GPU COMPUTING ARCHITECTURE

Introduced by NVIDIA in November 2006, the Tesla unified graphics and computing architecture[1,2] significantly extends the GPU beyond graphics—its massively multithreaded processor array becomes a highly efficient unified platform for *both* graphics and general-purpose parallel computing applications. By scaling the number of processors and memory partitions, the Tesla architecture spans a wide market range—from the high-performance enthusiast GeForce 8800 GPU and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs. Its computing features enable straightforward programming of the GPU cores in C with CUDA. Wide availability in laptops, desktops, workstations, and servers, coupled with C programmability and CUDA software, make the Tesla architecture the first ubiquitous supercomputing platform.

The Tesla architecture is built around a scalable array of multithreaded SMs (streaming multiprocessors). Current GPU implementations range from 768 to 12,288 concurrently executing threads. Transparent scaling across this wide range of available parallelism is a key design goal of both the GPU architecture and the CUDA programming model. Figure A shows a GPU with 14 SMs—a total of 112 SP (streaming processor) cores—interconnected with four external DRAM partitions. When a CUDA program on the host CPU invokes a kernel grid, the CWD (compute work distribution) unit enumerates the blocks of the grid and begins distributing them to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

An SM consists of eight scalar SP cores, two SFUs (special function units) for transcendentals, an MT IU (multithreaded instruction unit), and on-chip shared memory. The SM creates, manages, and executes up to 768 concurrent threads in hardware with zero scheduling overhead. It can execute as many as eight CUDA thread blocks concurrently, limited by thread and memory resources. The SM implements the CUDA __syncthreads() barrier synchronization intrinsic with a single instruction. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing a new thread to be created to compute each vertex, pixel, and data point.

To manage hundreds of threads running several different programs, the Tesla SM employs a new architecture we call

finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count.

THE CUDA PARADIGM

CUDA is a minimal extension of the C and C++ programming languages. The programmer writes a serial program that calls parallel *kernels*, which may be simple functions or full programs. A kernel executes in parallel across a set of parallel threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block. A *grid* is a set of thread blocks that may each be executed independently and thus may execute in parallel.

When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks

SIMT (single-instruction, multiple-thread).[3] The SM maps each thread to one SP scalar core, and each scalar thread executes independently with its own instruction address and register state. The SM SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. (This term originates from weaving, the first parallel thread technology.) Individual threads composing a

Continued on the next page

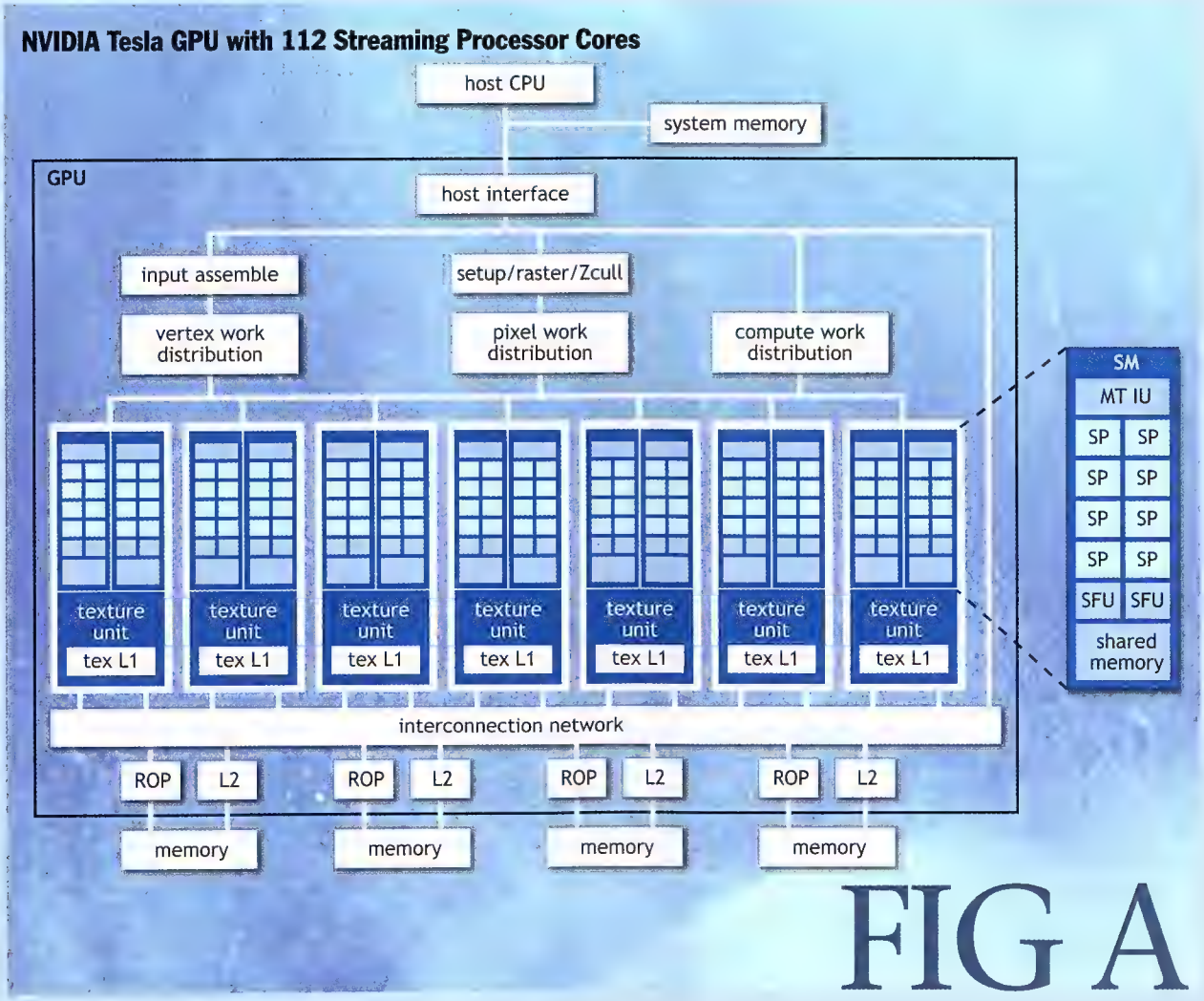


FIG A

Scalable Parallel PROGRAMMING with CUDA

making up the grid. Each thread is given a unique *thread ID* number threadIdx within its thread block, numbered 0, 1, 2, ..., blockDim-1, and each thread block is given a unique *block ID* number blockIdx within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have one, two, or three dimensions, accessed via .x, .y, and .z index fields.

As a very simple example of parallel programming, suppose that we are given two vectors x and y of n floating-point numbers each and that we wish to compute the result of $y \leftarrow ax + y$, for some scalar value a. This is the

SIMT warp start together at the same program address but are otherwise free to branch and execute independently. Each SM manages a pool of 24 warps of 32 threads per warp, a total of 768 threads.

Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. As a result, the Tesla-architecture GPUs are dramatically more efficient and flexible on branching code than previous-generation GPUs, as their 32-thread warps are much narrower than the SIMD (single-instruction multiple-data) width of prior GPUs.

SIMT architecture is akin to SIMD vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

A thread's variables typically reside in live registers. The 16KB SM shared memory has very low access latency and high bandwidth similar to an L1 cache; it holds CUDA per-block __shared__ variables for the active thread blocks. The SM provides load/store instructions to access CUDA __device__ variables in GPU external DRAM. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. Because global memory latency can be hundreds of processor clocks, CUDA programs copy data to shared memory when it must be accessed multiple times by a thread block. Tesla load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimizations. The large thread count in each SM, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM. The latest Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

CUDA applications perform well on Tesla-architecture GPUs because CUDA's parallelism, synchronization, shared memories, and hierarchy of thread groups map efficiently to features of the GPU architecture, and because CUDA expresses application parallelism well.

REFERENCES

1. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28(2).
2. Nickolls, J. 2007. NVIDIA GPU parallel computing architecture. In *IEEE Hot Chips* 19 (August 20), Stanford, CA; http://www.hotchips.org/archives/hc19/.
3. See reference 1.

so-called saxpy kernel defined by the BLAS (basic linear algebra subprograms) library. The code for performing this computation on both a serial processor and in parallel using CUDA is shown in figure 1.

The __global__ declaration specifier indicates that the procedure is a kernel entry point. CUDA programs launch parallel kernels with the extended function-call syntax

```
kernel<<<dimGrid, dimBlock>>>(... parameter list ...);
```

where dimGrid and dimBlock are three-element vectors of type dim3 that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively. Unspecified dimensions default to 1.

In the example, we launch a grid that assigns one thread to each element of the vectors and puts 256 threads in each block. Each thread computes an element index from its thread and block IDs and then performs the desired calculation on the corresponding vector elements. The serial and parallel versions of this code are strikingly similar. This represents a fairly common pattern. The serial code consists of a loop where each iteration is independent of all the others. Such loops can be mechanically transformed into parallel kernels: each loop iteration becomes an independent thread. By assigning a single thread to each output element, we avoid the need for any synchronization among threads when writing results to memory.

Computing y ← ax + y with a Serial Loop

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
   for(int i = 0; i<n; ++i)
      y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y ← ax + y in parallel using CUDA

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIG 1

The text of a CUDA kernel is simply a C function for one sequential thread. Thus, it is generally straightforward to write and is typically simpler than writing parallel code for vector operations. Parallelism is determined clearly and explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling, and termination are handled for the programmer by the underlying system. Indeed, a Tesla-architecture GPU performs all thread management directly in hardware. The threads of a block execute concurrently and may synchronize at a barrier by calling the __syncthreads() intrinsic. This guarantees that no thread participating in the barrier can proceed until all participating threads have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by participating threads before the barrier. Thus, threads in a block may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.

Since threads in a block may share local memory and synchronize via barriers, they will reside on the same physical processor or multiprocessor. The number of thread blocks can, however, greatly exceed the number of processors. This virtualizes the processing elements and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. This also allows the same CUDA program to scale to widely varying numbers of processor cores.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks execute independently. It must be possible to execute blocks in any order, in parallel or in series. Different blocks have no means of direct communication, although they may *coordinate* their activities using atomic memory operations on the global memory visible to all threads—by atomically incrementing queue pointers, for example.

This independence requirement allows thread blocks to be scheduled in any order across any number of cores, making the CUDA model scalable across an arbitrary

Scalable Parallel **PROGRAMMING** with **CUDA**

number of cores, as well as across a variety of parallel architectures. It also helps to avoid the possibility of deadlock.

An application may execute multiple grids either independently or dependently. Independent grids may execute concurrently given sufficient hardware resources. Dependent grids execute sequentially, with an implicit inter-kernel barrier between them, thus guaranteeing that all blocks of the first grid will complete before any block of the second dependent grid is launched.

Threads may access data from multiple memory spaces during their execution. Each thread has a private *local* memory. CUDA uses this memory for thread-private variables that do not fit in the thread's registers, as well as for stack frames and register spilling. Each thread block has a *shared* memory visible to all threads of the block that has the same lifetime as the block. Finally, all threads have access to the same *global* memory. Programs declare variables in shared and global memory with the __shared__ and __device__ type qualifiers. On a Tesla-architecture GPU, these memory spaces correspond to physically separate memories: per-block shared memory is a low-latency on-chip RAM, while global memory resides in the fast DRAM on the graphics board.

Shared memory is expected to be a low-latency memory near each processor, much like an L1 cache. It can, therefore, provide for high-performance communication and data sharing among the threads of a thread block. Since it has the same lifetime as its corresponding thread block, kernel code will typically initialize data in shared variables, compute using shared variables, and copy shared memory results to global memory. Thread blocks of sequentially dependent grids communicate via global memory, using it to read input and write results.

Figure 2 diagrams the nested levels of threads, thread blocks, and grids of thread blocks. It shows the corresponding levels of memory sharing: local, shared, and global memories for per-thread, per-thread-block, and per-application data sharing.

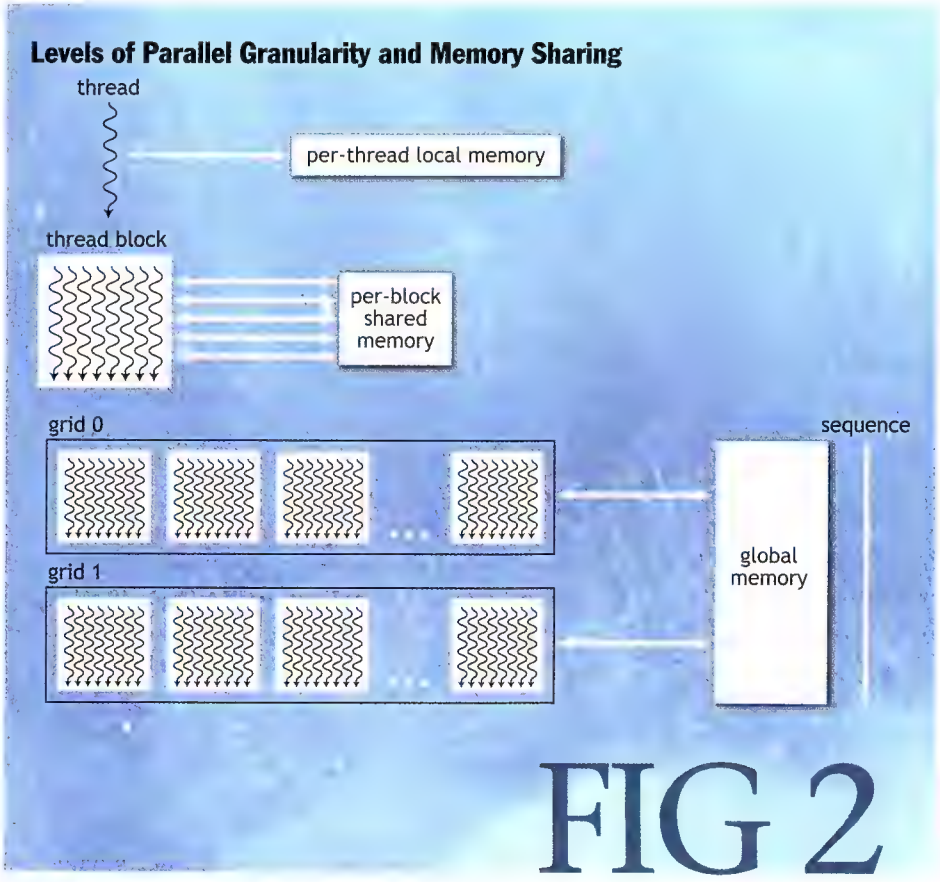


FIG 2

A program manages the global memory space visible to kernels through calls to the CUDA runtime, such as cudaMalloc() and cudaFree(). Kernels may execute on a physically separate device, as is the case when running kernels on the GPU. Consequently, the application must use cudaMemcpy() to copy data between the allocated space and the host system memory.

The CUDA programming model is similar in style to the familiar SPMD (single-program multiple-data) model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. CUDA, however, is more flexible than most real-

izations of SPMD, because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step. The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads.

Figure 3 shows an example of a SPMD-like CUDA code sequence. It first instantiates kernelF on a 2D grid of 3×2 blocks where each 2D thread block consists of 5×3 threads. It then instantiates kernelG on a 1D grid of four 1D thread blocks with six threads each. Because kernelG depends on the results of kernelF, they are separated by an inter-kernel synchronization barrier.

The concurrent threads of a thread block express fine-grained *data* and *thread* parallelism. The independent thread blocks of a grid express coarse-grained *data* parallelism. Independent grids express coarse-grained *task* parallelism. A *kernel* is simply C code for one thread of the hierarchy.

RESTRICTIONS

When developing CUDA programs, it is important to understand the ways in which the CUDA model is restricted, largely for reasons of efficiency. Threads and thread blocks may be created only by invoking a parallel kernel, not from within a parallel kernel. Together with the required independence of thread blocks, this makes it possible to execute CUDA programs with a simple scheduler that introduces minimal runtime overhead. In fact, the Tesla architecture implements *hardware* management and scheduling of threads and thread blocks.

Task parallelism can be expressed at the thread-block level, but blockwide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between thread blocks within the same kernel grid is not allowed—they must execute independently. Since CUDA requires that thread blocks be independent and allows blocks to be executed in any

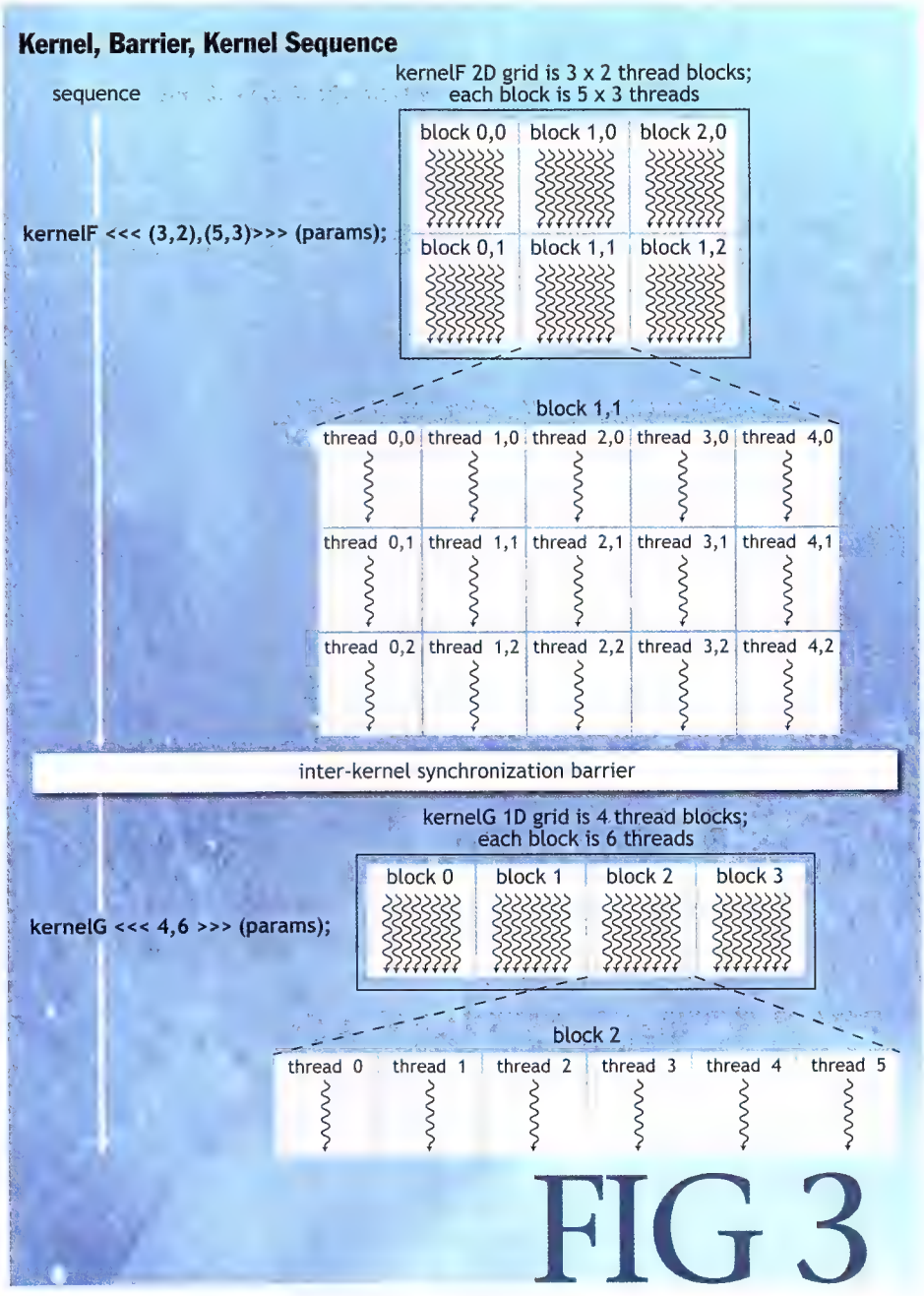


FIG 3

Scalable Parallel PROGRAMMING with CUDA

order, combining results generated by multiple blocks must in general be done by launching a second kernel on a new grid of thread blocks. However, multiple thread blocks can coordinate their work using atomic operations on global memory (e.g., to manage a data structure).

Recursive function calls are not allowed in CUDA kernels. Recursion is unattractive in a massively parallel kernel because providing stack space for the tens of thousands of threads that may be active would require substantial amounts of memory. Serial algorithms that are normally expressed using recursion, such as quicksort, are typically best implemented using nested data parallelism rather than explicit recursion.

To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU–GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects. Of course, problems large enough to need a GPU performance boost amortize the overhead better than small problems.

RELATED WORK

Although the first CUDA implementation targets NVIDIA GPUs, the CUDA abstractions are general and useful for programming multicore CPUs and scalable parallel systems. Coarse-grained thread blocks map naturally to separate processor cores, while fine-grained threads map to multiple-thread contexts, vector operations, and pipelined loops in each core. Stratton et al. have developed a prototype source-to-source translation framework that compiles CUDA programs for multicore CPUs by mapping a thread block to loops within a single CPU thread. They found that CUDA kernels compiled in this way perform and scale well.[4]

CUDA uses parallel kernels similar to recent GPGPU programming models, but differs by providing flexible thread creation, thread blocks, shared memory, global memory, and explicit synchronization. Streaming languages apply parallel kernels to data records from a stream. Applying a stream kernel to one record is analogous to executing a single CUDA kernel thread, but stream programs do not allow dependencies among kernel threads, and kernels communicate only via FIFO (first-in, first-out) streams. Brook for GPUs differentiates between FIFO input/output streams and random-access *gather* streams, and it supports parallel reductions. Brook is a good fit for earlier-generation GPUs with random access texture units and raster pixel operation units.[5]

Pthreads and Java provide fork-join parallelism but are not particularly convenient for data-parallel applications. OpenMP targets shared memory architectures with parallel execution constructs, including "parallel for" and teams of coarse-grained threads. Intel's C++ Threading Building Blocks provide similar features for multicore CPUs. MPI targets distributed memory systems and uses message passing rather than shared memory.

CUDA APPLICATION EXPERIENCE

The CUDA programming model extends the C language with a small number of additional parallel abstractions. Programmers who are comfortable developing in C can quickly begin writing CUDA programs.

In the relatively short period since the introduction of CUDA, a number of real-world parallel application codes have been developed using the CUDA model. These include FHD-spiral MRI reconstruction,[6] molecular dynamics,[7] and n-body astrophysics simulation.[8] Running on Tesla-architecture GPUs, these applications were able to achieve substantial speedups over alternative implementations running on serial CPUs: the MRI reconstruction was 263 times faster; the molecular dynamics code was 10–100 times faster; and the n-body simulation was 50–250 times faster. These large speedups are a result of the highly parallel nature of the Tesla architecture and its high memory bandwidth.

Compressed Sparse Row (CSR) Matrix

a. sample matrix *A*

$$\begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

b. CSR representation of matrix

	row 0	row 2	row 3
Av[7] = {	3 1	2 4 1	1 1 }
Aj[7] = {	0 2	1 2 3	0 3 }
Ap[5] = {	0 2	2 5 7	}

FIG 4

EXAMPLE: SPARSE MATRIX-VECTOR PRODUCT

A variety of parallel algorithms can be written in CUDA in a fairly straightforward manner, even when the data structures involved are not simple regular grids. SpMV (sparse matrix-vector multiplication) is a good example of an important numerical building block that can be parallelized quite directly using the abstractions provided by CUDA. The kernels we discuss here, when combined with the provided CUBLAS vector routines, make writing iterative solvers such as the conjugate gradient[9] method straightforward.

A sparse $n \times n$ matrix is one in which the number of nonzero entries m is only a small fraction of the total. Sparse matrix representations seek to store only the nonzero elements of a matrix. Since it is fairly typical that a sparse $n \times n$ matrix will contain only m=O(n) nonzero elements, this represents a substantial savings in storage space and processing time.

One of the most common representations for general unstructured sparse matrices is the CSR (compressed sparse row) representation. The m nonzero elements of the matrix A are stored in row-major order in an array Av. A second array Aj records the corresponding column index for each entry of Av. Finally, an array Ap of n+1

```
float multiply_row(unsigned int rowsize,
                   unsigned int *Aj, // column indices for row
                   float *Av,        // non-zero entries for row
                   float *x)         // the RHS vector
{
   float sum = 0;

   for(unsigned int column=0; column<rowsize; ++column)
       sum += Av[column] * x[Aj[column]];

   return sum;
}
```

FIG 5

```
void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
   for(unsigned int row=0; row<num_rows; ++row)
   {
       unsigned int row_begin = Ap[row];
       unsigned int row_end   = Ap[row+1];

       y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                             Av+row_begin, x);
   }
}
```

FIG 6

```
__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   float *x, float *y)
{
   unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

   if( row<num_rows )
   {
      unsigned int row_begin = Ap[row];
      unsigned int row_end   = Ap[row+1];

      y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                            Av+row_begin, x);
   }
}
```

FIG 7

Scalable Parallel **PROGRAMMING** with **CUDA**

elements records the extent of each row in the previous arrays; the entries for row *i* in Aj and Av extend from index Ap[i] up to, but not including, index Ap[i+1]. This implies that Ap[0] will always be 0 and Ap[n] will always be the number of nonzero elements in the matrix. Figure 4 shows an example of the CSR representation of a simple matrix.

Given a matrix *A* in CSR form, we can compute a single row of the product $y = Ax$ using the multiply_row() procedure shown in figure 5.

Computing the full product is then simply a matter of looping over all rows and computing the result for that row using multiply_row(), as shown in figure 6.

This algorithm can be translated into a parallel CUDA kernel quite easily. We simply spread the loop in csrmul_serial() over many parallel threads. Each thread will compute exactly one row of the output vector *y*. Figure 7 shows the code for this kernel. Note that it looks extremely similar to the serial loop used in the csrmul_serial() procedure. There are really only two points of difference. First, the row index is computed from the block and thread indices assigned to each thread. Second, we have a conditional that evaluates a row product only if the row index is within the bounds of the matrix (this is necessary since the number of rows *n* need not be a multiple of the block size used in launching the kernel).

Assuming that the matrix data structures have already been copied to the GPU device memory, launching this kernel will look like the code in figure 8.

The pattern that we see here is a common one. The original serial algorithm is a loop whose iterations are independent of each other. Such loops can be parallelized quite easily by simply assigning one or more iterations of the loop to each parallel thread. The programming model provided by CUDA makes expressing this type of parallelism particularly straightforward.

This general strategy of decomposing computations into blocks of independent work, and more specifically breaking up independent loop iterations, is not unique to CUDA. This is a common approach used in one form or another by various parallel programming systems, including OpenMP and Intel's Threading Building Blocks.

```
unsigned int blocksize = 128;  // or any size up to 512
unsigned int nblocks  = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);
```

FIG 8

```
__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                   float *Av, unsigned int num_rows,
                   const float *x, float *y)
{
  // Cache the rows of x[] corresponding to this block.
  __shared__ float cache[blocksize];

  unsigned int block_begin = blockIdx.x * blockDim.x;
  unsigned int block_end   = block_begin + blockDim.x;
  unsigned int row         = block_begin + threadIdx.x;

  // Fetch and cache our window of x[].
  if( row<num_rows)  cache[threadIdx.x] = x[row];
  __syncthreads();

  if( row<num_rows )
  {
    unsigned int row_begin = Ap[row];
    unsigned int row_end   = Ap[row+1];
    float sum = 0, x_j;

    for(unsigned int col=row_begin; col<row_end; ++col)
    {
      unsigned int j = Aj[col];

      // Fetch x_j from our cache when possible
      if( j>=block_begin && j<block_end )
        x_j = cache[j-block_begin];
      else
        x_j = x[j];

      sum += Av[col] * x_j;
    }

    y[row] = sum;
  }
}
```

FIG 9

CACHING IN SHARED MEMORY

The SpMV algorithms outlined here are fairly simplistic. We can make a number of optimizations in both the CPU and GPU codes that can improve performance, including loop unrolling, matrix reordering, and register blocking.[10] The parallel kernels can also be reimplemented in terms of data-parallel *scan* operations.[11]

One of the important architectural features exposed by CUDA is the presence of the per-block shared memory, a small on-chip memory with very low latency. Taking advantage of this memory can deliver substantial performance improvements. One common way of doing this is to use shared memory as a software-managed cache to hold frequently reused data, shown in figure 9.

In the context of sparse matrix multiplication, we observe that several rows of *A* may use a particular array element x[i]. In many common cases, and particularly when the matrix has been reordered, the rows using x[i] will be rows near row *i*. We can therefore implement a simple caching scheme and expect to achieve some performance benefit. The block of threads processing rows *i* through *j* will load x[i] through x[j] into its shared memory. We will unroll the multiply_row() loop and fetch elements of x from the cache whenever possible. The resulting code is shown in figure 9. Shared memory can also be used to make other optimizations, such as fetching Ap[row+1] from an adjacent thread rather than refetching it from memory.

Because the Tesla architecture provides an explicitly managed on-chip shared memory rather than an implicitly active hardware cache, it is fairly common to add this sort of optimization. Although this can impose some additional development burden on the programmer, it is relatively minor, and the potential performance benefits can be substantial. In the example shown in figure 9, even this fairly simple use of shared memory returns a roughly 20 percent performance improvement on representative matrices derived from 3D surface meshes. The availability of an explicitly managed memory in lieu of an implicit cache also has the advantage that caching and prefetching policies can be specifically tailored to the application needs.

EXAMPLE: PARALLEL REDUCTION

Suppose that we are given a sequence of *N* integers that must be combined in some fashion (e.g., a sum). This occurs in a variety of algorithms, linear algebra being a common example. On a serial processor, we would write a simple loop with a single accumulator variable to construct the sum of all elements in sequence. On a parallel machine, using a single accumulator variable would create a global serialization point and lead to very poor performance. A well-known solution to this problem is the so-called *parallel reduction* algorithm. Each parallel thread sums a fixed-length subsequence of the input. We then collect these partial sums together, by summing

```
__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory, padding
    // with 0 if N is not a multiple of blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Every thread now holds 1 input value in x[]
    //
    // Build summation tree over elements. See attached figure.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s)  x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 now holds the sum of all input values
    // to this block. Have it add that sum to the running total
    if( tid == 0 )  atomicAdd(total, x[tid]);
}
```

FIG 10

Scalable Parallel PROGRAMMING with CUDA

pairs of partial sums in parallel. Each step of this pair-wise summation cuts the number of partial sums in half and ultimately produces the final sum after $\log_2 N$ steps. Note that this implicitly builds a tree structure over the initial partial sums.

In the example shown in figure 10, each thread simply loads one element of the input sequence (i.e., it initially sums a subsequence of length one). At the end of the reduction, we want thread 0 to hold the sum of all elements initially loaded by the threads of its block. We can achieve this in parallel by summing values in a tree-like pattern. The loop in this kernel implicitly builds a summation tree over the input elements. The action of this loop for the simple case of a block of eight threads is illustrated in figure 11. The steps of the loop are shown as successive levels of the diagram and edges indicate from where partial sums are being read.

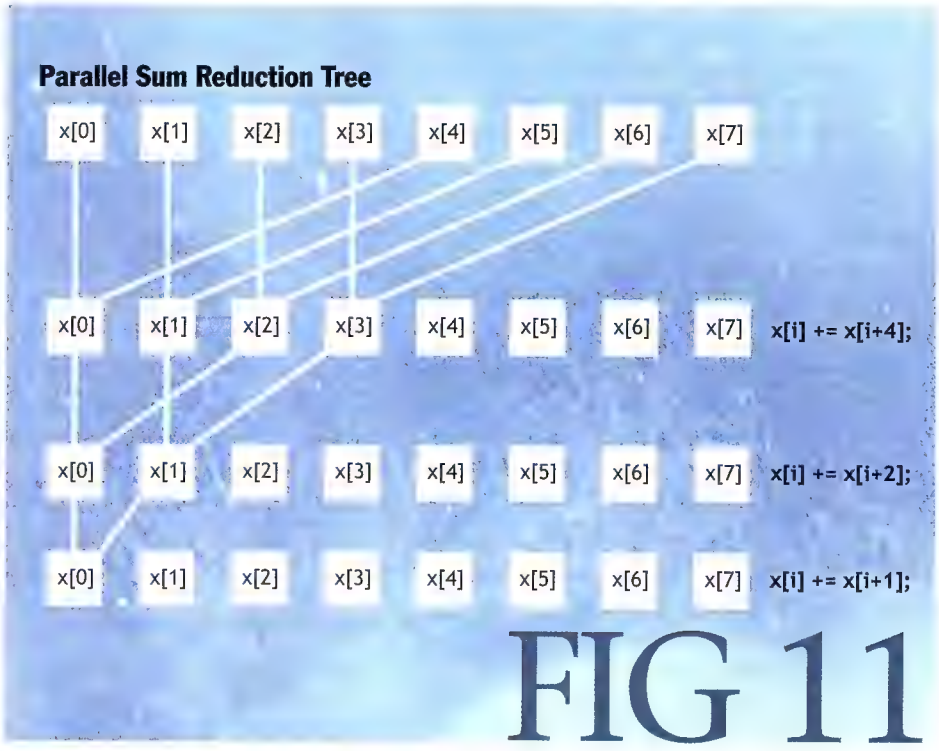


FIG 11

At the end of this loop, thread 0 holds the sum of all the values loaded by this block. If we want the final value of the location pointed to by total to contain the total of all elements in the array, we must combine the partial sums of all the blocks in the grid. One strategy would be to have each block write its partial sum into a second array and then launch the reduction kernel again, repeating the process until we had reduced the sequence to a single value. A more attractive alternative supported by the Tesla architecture is to use atomicAdd(), an efficient atomic read-modify-write primitive supported by the memory subsystem. This eliminates the need for additional temporary arrays and repeated kernel launches.

Parallel reduction is an essential primitive for parallel programming and highlights the importance of per-block shared memory and low-cost barriers in making cooperation among threads efficient. This degree of data shuffling among threads would be prohibitively expensive if done in off-chip global memory.

THE DEMOCRATIZATION OF PARALLEL PROGRAMMING

CUDA is a model for parallel programming that provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications. In fact, CUDA is an excellent programming environment for teaching parallel programming. The University of Virginia has used it as just a short, three-week module in an undergraduate computer architecture course, and students were able to write a correct k-means clustering program after just three lectures. The University of Illinois has successfully taught a semester-long parallel programming course using CUDA to a mix of computer science and non-computer science majors, with students obtaining impressive speedups on a variety of real applications, including the previously mentioned MRI reconstruction example.

CUDA is supported on NVIDIA GPUs with the Tesla unified graphics and computing architecture of the GeForce 8-series, recent Quadro, Tesla, and future GPUs.

The programming paradigm provided by CUDA has allowed developers to harness the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of 100 times or more on a variety of sophisticated applications.

The CUDA abstractions, however, are general and provide an excellent programming environment for multicore CPU chips. A prototype source-to-source translation framework developed at the University of Illinois compiles CUDA programs for multicore CPUs by mapping a parallel thread block to loops within a single physical thread. CUDA kernels compiled in this way exhibit excellent performance and scalability.[12]

Although CUDA was released less than a year ago, it is already the target of massive development activity—there are tens of thousands of CUDA developers. The combination of massive speedups, an intuitive programming environment, and affordable, ubiquitous hardware is rare in today's market. In short, CUDA represents a democratization of parallel programming. Q

REFERENCES

1. NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA.
2. NVIDIA. 2007. CUDA Programming Guide 1.1; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
3. Stratton, J.A., Stone, S. S., Hwu, W. W. 2008. M-CUDA: An efficient implementation of CUDA kernels on multicores. IMPACT Technical Report 08-01, University of Illinois at Urbana-Champaign, (February).
4. See reference 3.
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P. Brook for GPUs: Stream computing on graphics hardware. 2004. *Proceedings of SIGGRAPH* (August): 777-786; http://doi.acm.org/10.1145/1186562.1015800.
6. Stone, S.S., Yi, H., Hwu, W.W., Haldar, J.P., Sutton, B.P., Liang, Z.-P. 2007. How GPUs can improve the quality of magnetic resonance imaging. The First Workshop on General-Purpose Processing on Graphics Processing Units (October).
7. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K. 2007. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* 28(16): 2618–2640; http://dx.doi.org/10.1002/jcc.20829.
8. Nyland, L., Harris, M., Prins, J. 2007. Fast n-body simulation with CUDA. In *GPU Gems 3*. H. Nguyen, ed. Addison-Wesley.
9. Golub, G.H., and Van Loan, C.F. 1996. *Matrix Computations*, 3rd edition. Johns Hopkins University Press.
10. Buatois, L., Caumon, G., Lévy, B. 2007. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *Proceedings of the High-Performance Computation Conference (HPCC)*, Springer LNCS.
11. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D. 2007. Scan primitives for GPU computing. In *Proceedings of Graphics Hardware* (August): 97–106.
12. See Reference 3.

Links to the latest version of the CUDA development tools, documentation, code samples, and user discussion forums can be found at: http://www.nvidia.com/CUDA.

LOVE IT, HATE IT? LET US KNOW

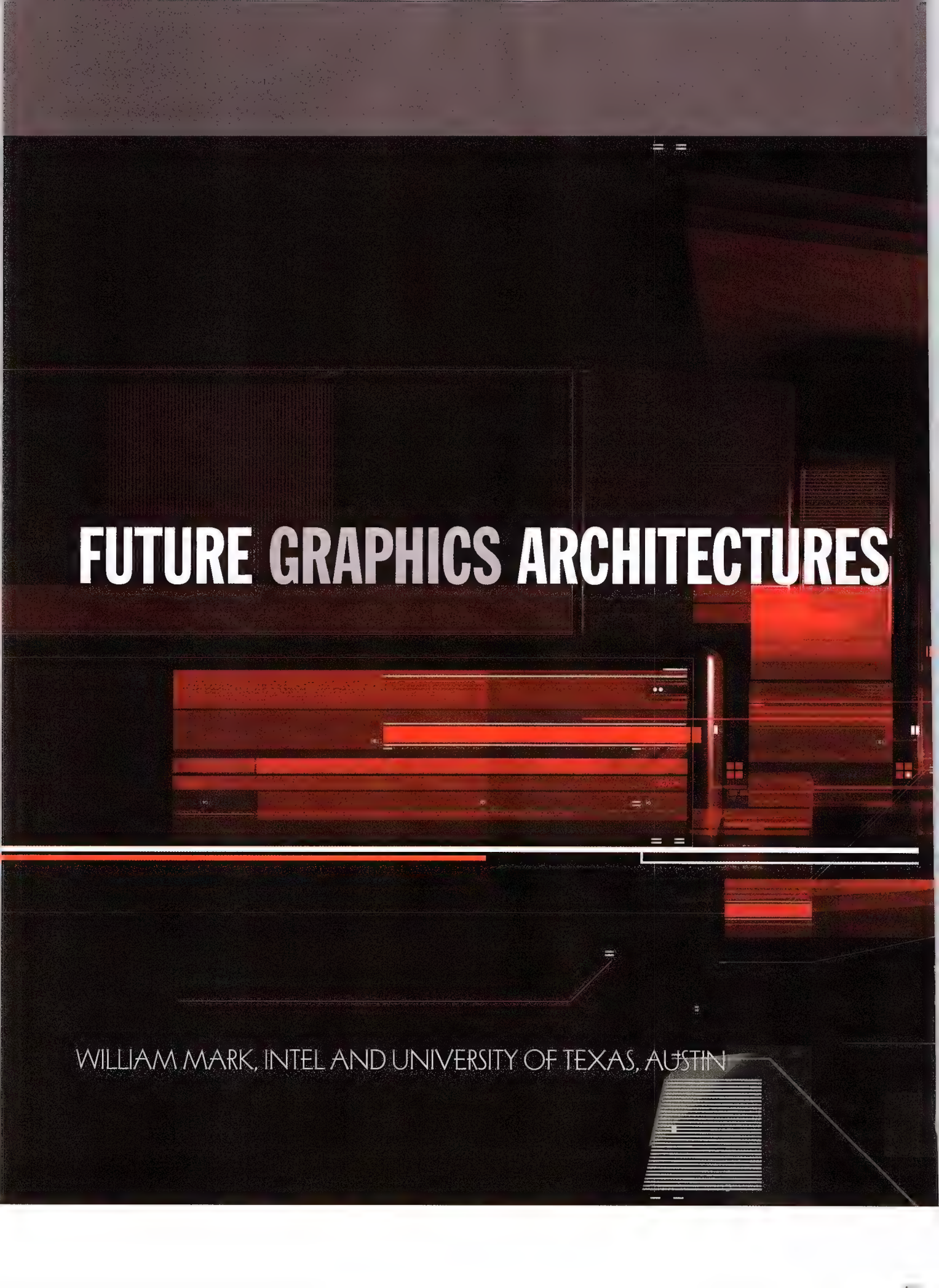
feedback@acmqueue.com or www.acmqueue.com/forums

JOHN NICKOLLS is director of architecture at NVIDIA for GPU computing. He was previously with Broadcom, Silicon Spice, Sun Microsystems, and was a cofounder of MasPar Computer. His interests include parallel processing systems, languages, and architectures. He has a B.S. in electrical engineering and computer science from the University of Illinois, and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

IAN BUCK works for NVIDIA as the GPU-Compute software manager. He completed his Ph.D. at the Stanford Graphics Lab in 2004. His thesis was titled "Stream Computing on Graphics Hardware," researching programming models and computing strategies for using graphics hardware as a general-purpose computing platform. His work included developing the Brook software tool chain for abstracting the GPU as a general-purpose streaming coprocessor.

MICHAEL GARLAND is a research scientist with NVIDIA Research. Prior to joining NVIDIA, he was an assistant professor in the department of computer science at the University of Illinois at Urbana-Champaign. He received Ph.D. and B.S. degrees from Carnegie Mellon University. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models.

KEVIN SKADRON is an associate professor in the department of computer science at the University of Virginia and is currently on sabbatical with NVIDIA Research. He received his Ph.D. from Princeton University and B.S. from Rice University. His research interests include power- and temperature-aware design, and manycore architecture and programming models. He is a senior member of the ACM.


FUTURE GRAPHICS ARCHITECTURES
WILLIAM MARK, INTEL AND UNIVERSITY OF TEXAS, AUSTIN

GPUs continue to evolve rapidly, but toward what?

Graphics architectures are in the midst of a major transition. In the past, these were specialized architectures designed to support a single rendering algorithm: the standard Z buffer. Realtime 3D graphics has now advanced to the point where the Z-buffer algorithm has serious shortcomings for generating the next generation of higher-quality visual effects demanded by games and other interactive 3D applications. There is also a desire to use the high computational capability of graphics architectures to support collision detection, approximate physics simulations, scene management, and simple artificial intelligence. In response to these forces, graphics architectures are evolving toward a general-purpose parallel-programming

FUTURE GRAPHICS ARCHITECTURES

model that will support a variety of image-synthesis algorithms, as well as nongraphics tasks.

This architectural transformation presents both opportunities and challenges. For hardware designers, the primary challenge is to balance the demand for greater programmability with the need to continue delivering high performance on traditional image-synthesis algorithms. Software developers have an opportunity to escape from the constraints of hardware-dictated image-synthesis algorithms so that almost any desired algorithm can be implemented, even those that have nothing to do with graphics. With this opportunity, however, comes the challenge of writing efficient, high-performance parallel software to run on the new graphics architectures. Writing such software is substantially more difficult than writing the single-threaded software that most developers are accustomed to, and it requires that programmers address challenges such as algorithm parallelization, load balancing, synchronization, and management of data locality.

The transformation of graphics hardware from a specialized architecture to a flexible high-throughput parallel architecture will have an impact far beyond the domain of computer graphics. For a variety of technical and business reasons, graphics architectures are likely to evolve into the dominant high-throughput "manycore" architectures of the future.

This article begins by describing the high-level forces that drive the evolution of realtime graphics systems, then moves on to some of the detailed technical trends in realtime graphics algorithms that are emerging in response to these high-level forces. Finally, it considers how future graphics architectures are expected to evolve to accommodate these changes in graphics algorithms and discusses the challenges that these architectures will present for software developers.

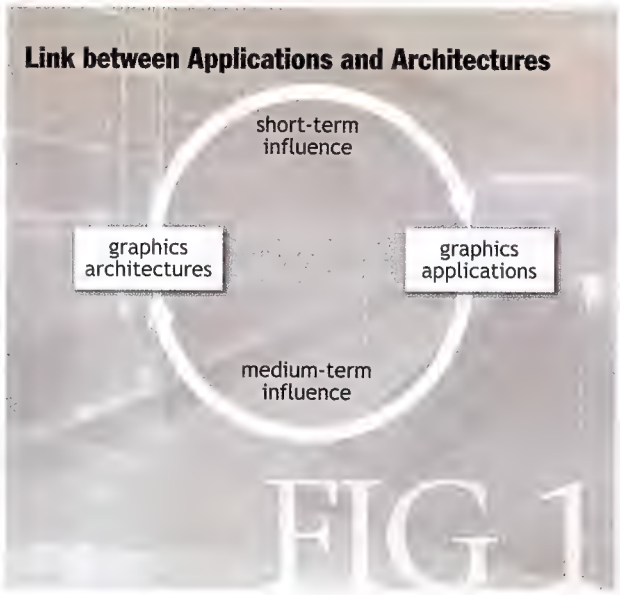
APPLICATIONS DRIVE EVOLUTION OF GRAPHICS ARCHITECTURES

To understand what form future graphics architectures are likely to take, we need to examine the forces that are driving the evolution of these architectures. As with any engineered artifact, graphics architectures are designed to deliver the maximum benefit to the end user within the fundamental technology constraints that determine what is affordable at a particular point in time. As VLSI (very large-scale integration) fabrication technology advances, the boundary of what is affordable changes, so that each generation of graphics architecture can provide additional capabilities at the same cost as the previous generation. Thus, the key high-level question is: What do we want these new capabilities to be?

Roughly speaking, graphics hardware is used for three purposes: 3D graphics, particularly entertainment applications (i.e., games); 2D desktop display, which used to be strictly 2D but now uses 3D capabilities for compositing desktops such as those found in Microsoft's Vista and Apple's Mac OS X; and video playback (i.e., decompression and display of streaming video and DVDs).

Although for most users desktop display and video playback are more important than 3D graphics, this article focuses on the needs of 3D graphics because these applications, with their significant demands for performance and functionality, have been the strongest force driving the evolution of graphics architectures.

Designing a graphics system for future 3D entertainment applications is particularly tricky because at a technical level the goals are ill defined. It is currently not possible to compute an image of the ideal quality at realtime frame rates, as evidenced by the fact that the images in computer-generated movies are of higher quality than those in computer games. Thus, designers must make approximations to the ideal computation. There are an enormous variety of possible approximations to choose



from, each of which introduces a different kind of visual artifact in the image, and each of which uses different algorithms that may in turn run best on different architectures. In essence, the system design problem is reduced to the ill-specified problem of which system (software and hardware) produces the best-quality game images for a specific cost. Figure 1 illustrates this problem. In practice, there are also other constraints, such as backward compatibility and a desire to build systems that facilitate content creation.

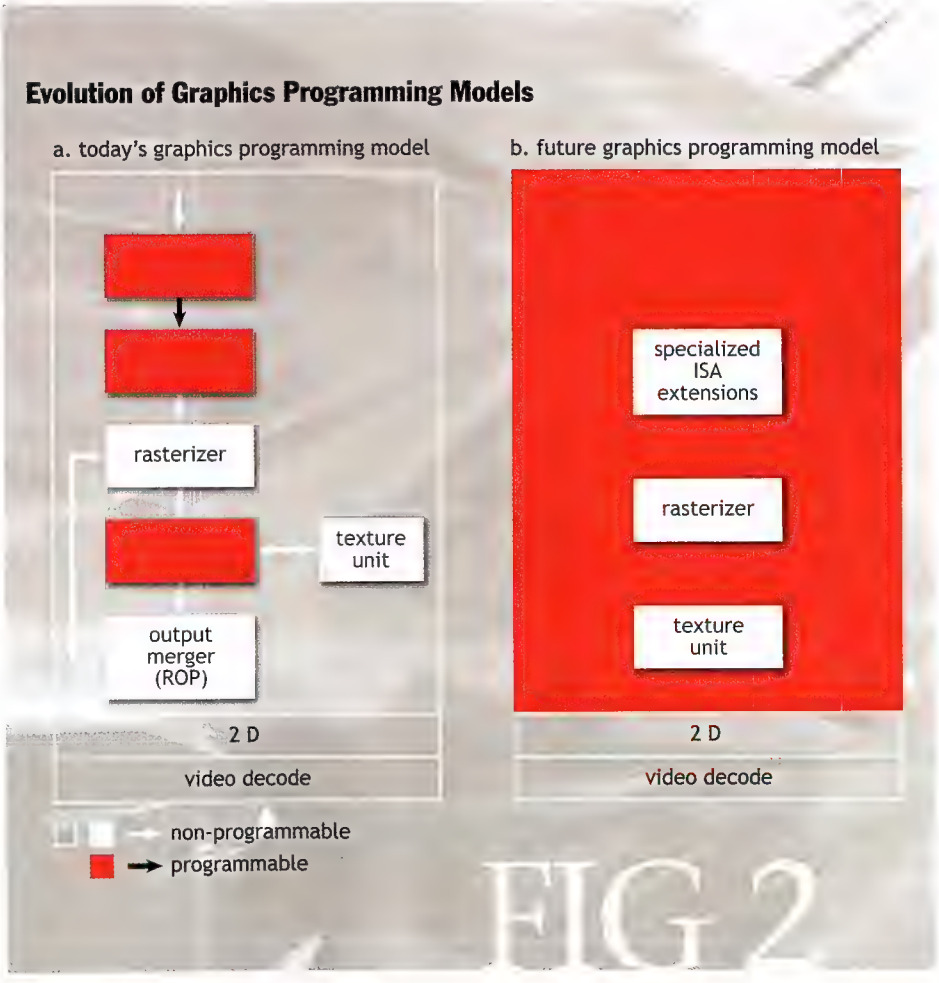
As VLSI technology advances with time, the system designer is provided with more transistors. If we assume that the frame rate is fixed at 60 Hz, the additional computational capability provided by these transistors can be used in three fundamental ways: increasing the screen resolution; increasing the scene detail (polygon count or material shader complexity); and changing the overall approximations, by changing the basic rendering algorithm or specific components of it.

Looking back at the past six years, we can see these forces at work. Games have adopted programmable shaders that allow sophisticated modeling of materials and multipass techniques that approximate shadows, reflections, and other effects. Graphics architectures have enabled these changes through the addition of programmable vertex and fragment units, as well as more flexibility in how data moves between stages in the graphics pipeline.

Current graphics processors use the programming model illustrated in figure 2a. This model supports the traditional Z-buffer algorithm and is organized around a predefined pipeline structure that is only partially reconfigurable by the application.[1] The predefined pipeline structure employs specialized hardware for the Z-buffer algorithm (in particular for polygon rasterization and Z-buffer read-modify-write operations), as well as for other operations such as the thread scheduling needed by the programmable stages.

Many of the individual pipeline stages are programmable (to support programmable material shading computations in particular), with all of the programmable stages multiplexed onto a single set of homogeneous programmable hardware processors. The programs executing within these pipeline stages, however, are heavily restricted in how they can communicate with each other and in how—if at all—they can access the global shared memory. This programming model provides high performance for the computations it is designed to support, but makes it difficult to support other computations efficiently.

It is important to realize that modern game applications fundamentally *require* programmability in the graphics hardware. This is because the real world contains an enormous variety of materials (wood, metal, glass, skin, fur, ...), and the only reasonable way to specify the



FUTURE GRAPHICS ARCHITECTURES

interactions of these materials with light is to use a different program for each material.

This situation is very different from that found for other high-performance tasks, such as video decode, which does not inherently require programmable hardware; one could design fixed-function hardware sufficient to support the standard video formats without any programmability at all. As a practical matter most video-decode hardware does include some programmable units, but this is an implementation choice, not a fundamental requirement. This need for programmability by 3D graphics applications makes graphics architectures uniquely well positioned to evolve into more general high-throughput parallel computer architectures that handle tasks beyond graphics.

LIMITS OF THE TRADITIONAL Z-BUFFER GRAPHICS PIPELINE

The Z-buffer graphics pipeline with programmable shading that is used as the basis of today's graphics architectures makes certain fundamental approximations and assumptions that impose a practical upper limit on the image quality. For example, a Z buffer cannot efficiently determine if two arbitrarily chosen points are visible from each other, as is needed for many advanced visual effects. A ray tracer, on the other hand, can efficiently make this determination. For this reason, computer-generated movies use rendering techniques such as ray-tracing algorithms and the Reyes (renders everything you ever saw) algorithm[2] that are more sophisticated than the standard Z-buffer graphics pipeline.

Over the past few years, it has become clear that the next frontier for improved visual quality in realtime 3D graphics will involve modeling lighting and complex illumination effects more realistically (but not necessarily photo-realistically) so as to produce images that are closer in quality to those of computer-generated movies. These effects include hard-edged shadows (from small lights), soft-edged shadows (from large lights), reflections from water, and approximations to more complex effects such as diffuse lighting interactions that dominate most interior environments. There is also a desire to model effects such as motion blur and to use higher-quality anti-aliasing techniques. Most of these effects are challenging to produce with the traditional Z-buffer graphics pipeline.

Modern game engines (e.g., Unreal Engine 3, CryEngine 2) have begun to support some of these effects using today's graphics hardware, but with significant limitations. For example, Unreal Engine 3 uses four different shadow algorithms, because no one algorithm provides an acceptable combination of performance and image quality in all situations. This problem is a result of limitations on the visibility queries that are supported by the traditional Z-buffer pipeline. Furthermore, it is common for different effects such as shadows and partial transparency to be mutually incompatible (e.g., partially transparent objects cast shadows as if they were fully opaque objects). This lack of algorithmic robustness and generality is a problem for both game-engine programmers and for the artists who create the game content. These limitations can also be viewed as violations of important principles of good system design such as abstraction (a capability should work for all relevant cases) and orthogonality (different capabilities should not interact in unexpected ways).

The underlying problem is that the traditional Z-buffer graphics pipeline was designed to compute visibility (i.e., the first surface hit) for regularly spaced rays originating at a single point (see figure 3a), but effects such as hard-edged shadows, soft-edged shadows, reflections, and diffuse lighting interactions all require more general visibility computations. In particular, reflections and diffuse lighting interactions require the ability to compute visible surfaces efficiently along rays with a variety of origins and directions (figure 3d). These types of visibility queries cannot be performed efficiently with the traditional graphics pipeline, but VLSI technology now provides enough transistors to support more sophisticated realtime visibility algorithms that can perform these queries efficiently. These transistors, however, must be organized into an architecture that can efficiently support the more sophisticated visibility algorithms.

Since the Z-buffer graphics pipeline is ill suited for producing the desired effects, the natural solution is to design graphics systems around more powerful visibility algorithms. Figure 3 provides an overview of some of these algorithms. I believe that these more powerful visibility algorithms will be gradually adopted over the next few years in response to the inadequacies of the standard Z buffer, although there is substantial debate in the graphics community as to how rapidly this change will occur. In particular, algorithms such as ray tracing

are likely to be adopted much more rapidly in realtime graphics than they were in movie rendering, because realtime graphics does not permit the hand-tweaking of lighting for every shot that is common in movie rendering.

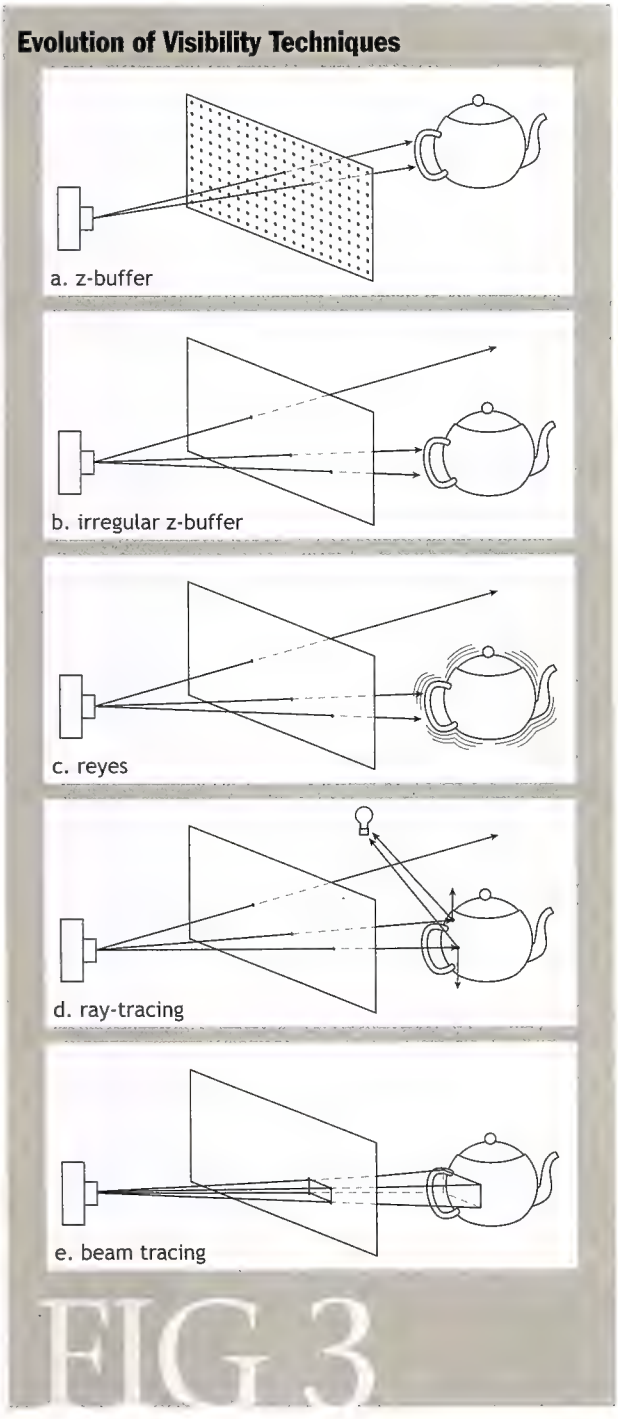
THE ARGUMENT FOR GENERAL-PURPOSE GRAPHICS HARDWARE

Given the desire to support more powerful visibility algorithms, graphics architects could take several approaches. Should the new visibility techniques be implemented in some kind of specialized hardware (like today's Z-buffer visibility computations), or should they be implemented in software on a flexible parallel architecture? I believe that a flexible parallel architecture is the best choice, because it supports the following software capabilities:

Mixing visibility techniques. Flexible hardware supports multiple visibility algorithms, ranging from the traditional Z buffer to ray tracing and beam tracing. Each application can choose the best algorithm(s) for its needs. The more sophisticated of these visibility algorithms require the ability to build and traverse irregular data structures such as KD-trees, which demands a more flexible parallel programming model than that used by today's GPUs.

Application-tailored approximations. Rendering images at realtime frame rates requires making mathematical approximations (e.g., for particular lighting effects), but the variety of possible approximations is enormous. Often, different approximations use very different overall rendering algorithms and have very different performance characteristics. Since the best approximation and algorithm vary from application to application and sometimes even within an application, an architecture that allows the application to choose its approximations can provide far greater efficiency for the overall rendering task than an architecture that lacks this flexibility.

Integration of rendering with scene management. Traditionally, realtime graphics systems have used one set of data structures to represent the persistent state of the scene (e.g., object positions, velocities, and groupings) and a different set of data structures to compute visibility. The two sets of data structures are on opposite sides of an intervening API such as DirectX or OpenGL. For every frame, all of the visible geometry is transferred across this API. In a Z-buffer system this approach works because it is relatively straightforward to determine which geometry might be visible. In a ray-tracing system, however, this approach does not work very well, and it is desirable to integrate the two sets of data structures more tightly, with both residing on the graphics processor (figure 4). It is also desirable to change the traditional layering of APIs so that the game engine takes over most of the low-level rendering tasks currently handled by graphics hardware (figure 5). A highly programmable architecture makes it much easier to do this integration while still preserving flexibility for the application to maintain the persistent



FUTURE GRAPHICS ARCHITECTURES

data structures in the most efficient manner. It also allows scene management computations to be performed on the high-performance graphics hardware, eliminating a bottleneck on the CPU.

Support for game physics and AI. A flexible parallel architecture can easily support computations such as collision detection, fluid dynamics simulations (e.g., for explosions), and artificial intelligence for game play. It also allows these computations to be tightly integrated with the rendering computation.

Rapid innovation. Software can be changed more rapidly than hardware, so a flexible parallel architecture that uses software to express its graphics algorithms enables more rapid innovation than traditional designs.

The best choice for the system as a whole is to use flexible parallel hardware that permits software to use aggressive algorithmic specialization and optimization, rather than to use specialized parallel hardware that mandates a particular algorithm.

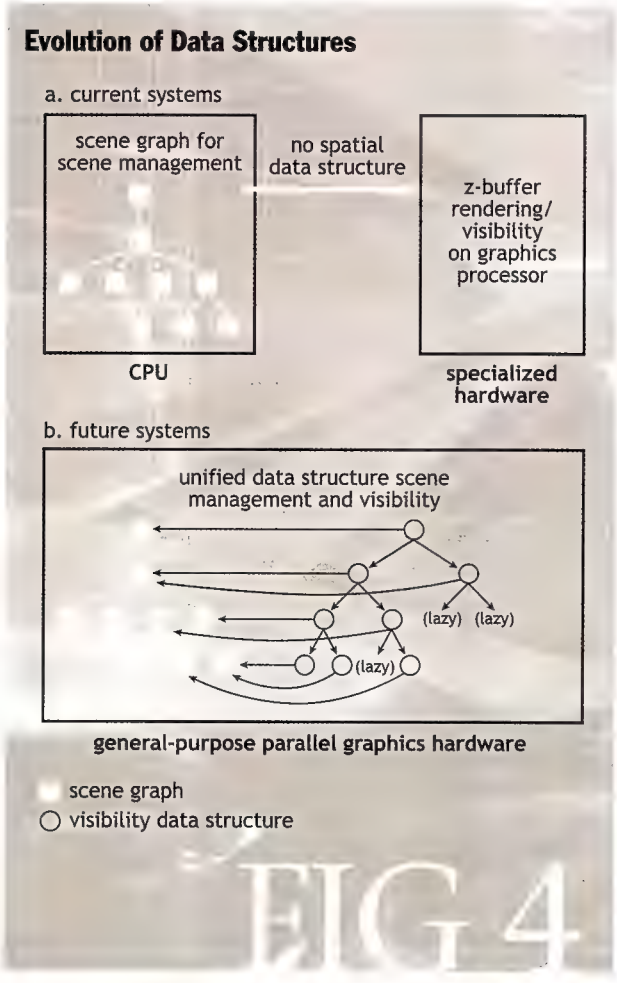


FIG 4

PROGRAMMING MODEL

When I say that future graphics architectures are likely to support an extremely flexible parallel programming model, what do I mean? There is considerable debate within the graphics hardware community as to the specific programming model that graphics architectures should adopt in the near future. I expect that in the short term each of the major graphics hardware companies will take a somewhat different path. There are a variety of reasons for this diversity: different emphasis placed on adding new capabilities versus improving performance of the old programming models; fundamental philosophical differences in tackling the parallel programming problem; and the desire by some companies to evolve existing designs incrementally.

In the longer term (five years or so), the programming models will probably converge, but there is not yet a consensus on what such a converged programming model would look like. This section presents some of the key issues that today's graphics architects face, as well as thoughts on what a converged future programming model could look like and the challenges that it

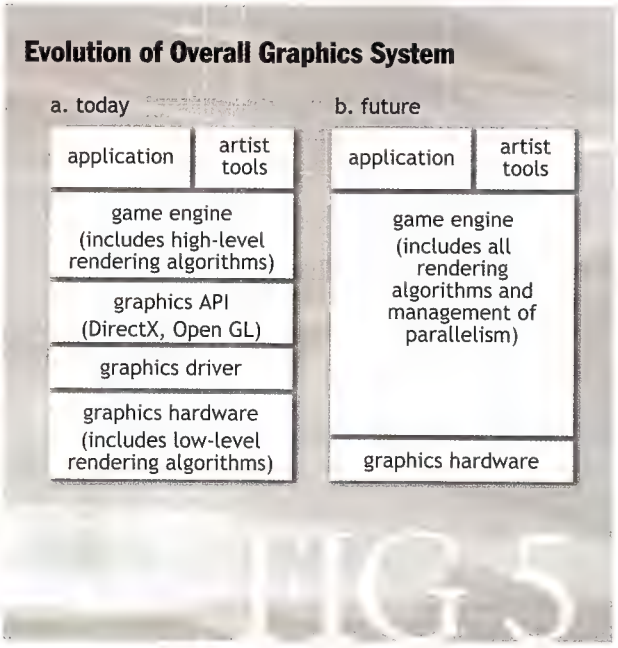


FIG 5

will present for programmers. Most of the programming challenges discussed here will be applicable to all future graphics architectures, even those that are somewhat different from the one I am expecting.

END OF THE HARDWARE-DEFINED PIPELINE

Graphics processors will evolve toward a programming model similar to that illustrated in figure 2b. User-written software specifies the overall structure of the computation, expressed in an extremely flexible parallel programming model similar to that used to program today's multicore CPUs. The user-written software may optionally use specialized hardware to accelerate specific tasks such as texture mapping. The specialized hardware may be accessed via a combination of instructions in the ISA (instruction set architecture), special memory-mapped registers, and special inter-processor messages.

The latest generation of GPUs (graphics processing units) from NVIDIA and AMD have already taken a significant step toward this future graphics programming model by supporting a separate programming model for nongraphics computations that is more flexible than the programming model used for graphics. This second programming model is an assembly-level parallel-programming model with some capabilities for fine-grained synchronization and data sharing across hardware threads. NVIDIA calls its model PTX (Parallel Thread Execution), and AMD's is known as CTM (Close to Metal). Note that NVIDIA's C-like CUDA language (see "Scalable Parallel Programming with CUDA" in this issue) is a layer on top of the assembly-level PTX. It is important to realize, however, that PTX and CTM have some significant limitations compared with traditional general-purpose parallel programming models. PTX and CTM are still fairly restrictive, especially in their memory and concurrency models.

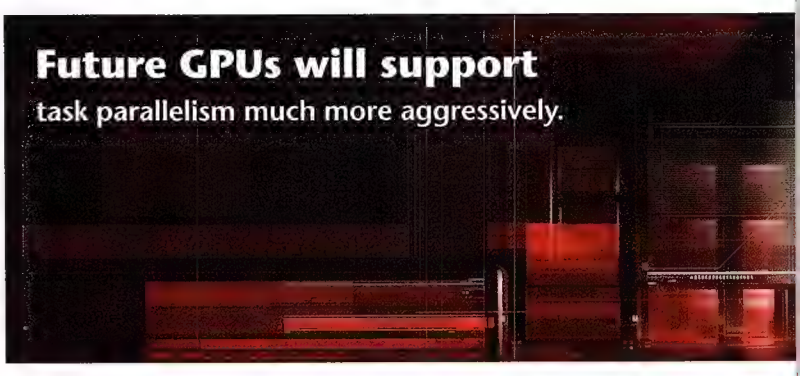
These limitations become obvious when comparing PTX and CTM with the programming models supported by other single-chip highly parallel processors, such as Sun's Niagara server chips. I believe that the programming model of future graphics architectures will be substantially more flexible than PTX and CTM.

TASK PARALLELISM AND MULTITHREADING

The parallelism supported by current GPUs primarily takes the form of data parallelism—that is, the GPU operates simultaneously on many data elements (such as vertices or pixels or elements in an array). In contrast, task parallelism is not supported well, except for the specific case of concurrent processing of pixels and vertices. Since better support for task parallelism is necessary to support user-defined rendering pipelines efficiently, I expect that future GPUs will support task parallelism much more aggressively. In particular, multiple tasks will be able to execute asynchronously from each other and from the CPU, and will be able to communicate and synchronize with each other. These changes will require a substantially more sophisticated software runtime environment than the one used for today's GPUs and will introduce significant complexity into the hardware/software interactions for thread management.

As with today's GPUs and Sun's Niagara processor, each core will use hardware multithreading,[3] possibly augmented by additional software multithreading along the lines of that used by programmers of the Cell architecture. This multithreading serves two purposes:

- First, it allows the core to remain fully utilized even if each individual instruction has a pipeline latency of several cycles—the core just executes an instruction from another thread.
- Second, it allows the core to remain fully utilized even if one or more of the threads on the core stalls because of an off-chip DRAM access such as those that occur when fetching data from a texture. Programmers will face the challenge of exposing parallelism for multiple cores *and* for multiple threads on each core. This challenge is already starting to appear with programming models such as NVIDIA's CUDA.



SIMD EXECUTION WITHIN EACH CORE

An important concern in the design of graphics hardware is obtaining the maximum possible performance using a fixed number of transistors on a chip. If one instruction cache/fetch/decode unit can be shared among several arithmetic units, the die area and power requirements of the hardware are reduced, as compared with a design that has one instruction unit per arithmetic unit. That

FUTURE GRAPHICS ARCHITECTURES

is, a SIMD (single instruction, multiple data) execution model increases efficiency as long as most of the elements in the SIMD vectors are kept active most of the time. A SIMD execution model also provides a simple form of fine-grained synchronization that helps to ensure that memory accesses have good locality.

Current graphics hardware uses a SIMD execution model, although it is sometimes hidden from the programmer behind a scalar programming interface as in NVIDIA's hardware. One area of ongoing debate and change is likely to be in the underlying hardware SIMD width; there is a tension between the efficiency gained for regular computations as SIMD width increases and the efficiency gained for irregular computations as SIMD width decreases. NVIDIA GPUs (GeForce 8000 and 9000 series) have an effective SIMD width of 32, but the trend has been for the SIMD width of GPUs to decrease to improve the efficiency of algorithms with irregular control flow.

There is also debate about how to expose the SIMD execution model. It can be directly exposed to the programmer with register-SIMD instructions, as is done with x86 SSE instructions, or it may be nominally hidden from the programmer behind a scalar programming model, as is the case with NVIDIA's GeForce 9000 series. If the SIMD execution model is hidden, the conversion from the scalar programming model to the SIMD hardware may be performed by either the hardware (as in the GeForce 9000 series) or a compiler or some combination of the two. Regardless of which strategy is used, programmers who are concerned with performance will need to be aware of the underlying SIMD execution model and width.

SMALL AMOUNTS OF LOCAL STORAGE

One of the most important differences between GPUs and CPUs is that GPUs devote a greater fraction of their transistors to arithmetic units, whereas CPUs devote a greater fraction of their transistors to cache. This difference is one of the primary reasons that the peak performance of a GPU is much higher than that of a CPU.

I expect that this difference will continue in the future. The impact on programmers will be significant: although the overall programming model of future GPUs will become much closer to that of today's CPUs, programmers will need to manage data locality much more carefully on future GPUs than they do on today's CPUs.

This problem is made even more challenging by multithreading; if there are N threads on each core, the amount of local storage per thread per core is effectively 1/N of the core's total local storage. This issue can be mitigated if the N threads on a core are sharing a working set, but to do this the programmer must think of the N threads as being closely coupled to each other. Similarly, programmers will have to think about how to share a working set across threads on different cores.

These considerations are already becoming apparent with CUDA. The constraints are likely to be frustrating to programmers who are accustomed to the large caches of CPUs, but they need to realize that extra local storage would come at the cost of fewer ALUs (arithmetic logic units), and they will need to work closely with hardware designers to determine the optimum balance between cache and ALUs.

CACHE-COHERENT SHARED MEMORY

The most important aspect of any parallel architecture is its overall memory and communication model. To illustrate the importance of this aspect of the design, consider four (of many) possible alternatives (of course, hybrids and enhancements of these models are possible):

- A message-passing architecture, in which each processor core has its own memory space and all communication occurs through explicit message passing. Most large-scale supercomputers (those with 100-plus processors) use this model.
- An architecture such as the Sony/Toshiba/IBM Cell with a noncached, noncoherent shared memory. In such an architecture, all transfers of data between a core's small private memory and the global memory must be orchestrated through explicit memory-transfer commands.
- An architecture such as NVIDIA's GeForce 8800 with what amounts to a minimally cached, noncoherent shared memory, with support for load/store to this memory.
- An architecture such as modern multicore CPUs, with cached, coherent shared memory. In such architectures, hardware mechanisms manage transfer of data between cache and main memory and ensure that data in caches of different processors remains consistent.

There is considerable debate within the graphics architecture community as to which memory and communication model would be best for future architectures, and

in the near term different hardware vendors are taking different approaches. Software programmers should think carefully about these issues so that they are prepared to influence the debate.

Which approach is most likely to dominate in the medium to long term? I have previously argued that the trend in rendering algorithms is toward those that build and traverse irregular data structures. These irregular data structures allow algorithms to adapt to the scene geometry and the current viewpoint. Explicitly managing all data locality for these algorithms is painful, especially if multiple cores share a read/write data structure. In my experience, it is easier to develop these algorithms on a cache-coherent architecture, even if achieving optimal

performance often still requires thinking very carefully about the communication and memory-access patterns of the performance-critical kernels.

For these and other reasons too detailed to discuss here, I believe that future graphics architectures will efficiently support a cache-coherent memory model, and that any architecture lacking these capabilities will be a second choice at best for programmers who are developing innovative rendering techniques. Sun's Niagara architecture provides a good preview of the kind of memory and threading model that I anticipate for future GPUs. I also expect, however, that cache-coherent graphics architectures will include a variety of mechanisms that provide the programmer with explicit control over communication and memory access, such as streaming loads that bypass the cache.

FINE-GRAINED SPECIALIZATION

The desire to support greater algorithmic diversity will drive future graphics architectures toward greater flexibility and generality, but specialization will still be used where it provides a sufficiently large benefit for the majority of applications. Most of this specialization will be at a fine granularity, used to accelerate specific operations, in contrast to the coarse, monolithic granularity used to dictate the overall structure of the algorithms executed on the hardware in the past.

In particular, I expect the following specialization will continue to exist for graphics architectures:

Texture hardware. Texture addressing and filtering operations use low-precision (typically 16-bit) values that are decompressed on the fly from a compressed representation stored in memory. The amount of data accessed is large and requires multithreading to deal effectively with cache misses. These operations are a significant fraction of the overall rendering cost and benefit enormously from specialized hardware.

Specialized floating-point operations. Rendering makes heavy use of floating-point square-root and reciprocal operations. Current graphics hardware provides high-performance instructions for these operations, as well as other operations used for shading such as swizzling and trigonometric functions. Future graphics hardware will need to do the same.

Video playback and desktop compositing. Video playback and 2D and 2.5D desktop window operations benefit significantly from specialized hardware. Specialization of these operations is especially important for power efficiency. I anticipate that much of this hardware will follow the traditional coarse-grained monolithic fixed-function model and thus will not be useful for user-written 3D graphics programs.

Current graphics hardware also includes specialized hardware to assist with triangle rasterization, but I expect that this task will be taken over by software within a few years. The reason is that rasterization is gradually becoming a smaller fraction of total rendering costs, so the penalty for implementing it in software is decreasing. This trend will accelerate as more sophisticated visibility algorithms supplement or replace the Z buffer.

As graphics software switches to more powerful visibility algorithms such as ray tracing, it may become clear that certain operations represent a sufficiently large portion of the total computation cost that hardware acceleration would be justified. For example, future architectures could include specialized instructions to accelerate the data-structure traversal operations used by ray tracing.

THE CHALLENGE FOR GRAPHICS ARCHITECTS

At a high level, the key challenge facing future graphics architectures is to strike the best balance between the desire to provide high performance on existing graphics algorithms and the desire to provide the flexibility needed to support new algorithms with high perfor-

FUTURE GRAPHICS ARCHITECTURES

mance, including nongraphics algorithms and the next generation of more capable and sophisticated graphics algorithms. I believe that the opportunity for improved visual quality and robustness provided by more sophisticated graphics algorithms will cause the transition to more flexible architectures to happen relatively rapidly, an opinion that remains a matter of debate within the graphics architecture community.

THE FUTURE OF GRAPHICS ARCHITECTURES

In the past, graphics architectures defined the algorithms used for rendering and their performance. In the future, graphics architectures will cease to define the rendering algorithms and will simply set the performance and power efficiency limits within which software developers may do whatever they want.

For the programmer, future graphics architectures are likely to be very similar to today's multicore CPU architectures, but with greater SIMD instruction widths and the availability of specialized instructions and processing units for some operations. Like today's Niagara processor, however, the amount of cache per processor core will be relatively small. To achieve peak performance, programmers will have to think more carefully about memory-access patterns and data-structure sizes than they have been accustomed to with the large caches of modern CPUs.

Future graphics architectures will enable a golden age of innovation in graphics; I expect that over the next few years we will see the development of a variety of new rendering algorithms that are more efficient and more capable than the ones used in the past. For computer games, these architectures will allow game logic, physics simulation, and AI to be more tightly integrated with rendering than before. For data-visualization applications, these architectures will allow tight integration of domain-specific data analysis with the rendering computations used to display the results of this analysis. The general-purpose nature of these architectures combined with the low cost enabled by their high-volume market will also cause them to become the preferred platform for almost all high-performance floating-point computations. Q

ACKNOWLEDGMENTS AND FURTHER READING

Don Fussell, Kurt Akeley, Matt Pharr, Pat Hanrahan, Mark Horowitz, Stephen Junkins, and several graphics hardware architects contributed directly and indirectly to the ideas in this article through many fun and productive discussions. More details about many of the ideas discussed in this article can be found in another article I wrote with Don Fussell in 2005.[4] The tendency of graphics hardware to become increasingly general until the temptation emerges to incorporate new specialized units has existed for a long time and was described in 1968 as the "wheel of reincarnation" by Myer and Sutherland.[5] The fundamental need for programmability in realtime graphics hardware, however, is much more important now than it was then.

REFERENCES

1. Blythe, D. 2006. The Direct3D 10 system. In *ACM SIGGRAPH 2006 Papers*: 724–734.
2. Cook, R.L., Carpenter, L., Catmull, E. 1987. The Reyes image rendering architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH)*: 95–102.
3. Laudon, J., Gupta, A., Horowitz, M. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*: 308–318.
4. Mark, W., Fussell, D. 2005. Real-time rendering systems in 2010. Technical Report 05-18, University of Texas.
5. Myer, T.H., Sutherland, I.E. 1968. On the design of display processors. *Communications of the ACM*, 11(6): 410–414.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BILL MARK leads Intel's advanced graphics research lab. He is on leave from the University of Texas at Austin, where until January 2008 he led a research group that investigated future graphics algorithms and architectures. In 2001-2002 he was the technical leader of the team at NVIDIA that co-designed (with Microsoft) the Cg language for programmable graphics hardware and developed the first release of the NVIDIA Cg compiler. His research interests focus on systems and hardware architectures for realtime computer graphics and on the opportunity to extend these systems to support more general parallel computation and a broader range of graphics algorithms, including interactive ray tracing.

Essential RenderMan

Ian Stephenson, Springer-Verlag New York, Inc., 2007, $39.95, ISBN: 1846283442

RenderMan, a rendering API associated with film production, is an essential tool for creating many of the effects and images in recent animated films (such as *Monsters Inc.* and *Finding Nemo*). *Essential RenderMan* is a self-paced tutorial that facilitates the understanding of many of the features of the API. The tone of the book is set in the preface, where author Ian Stephenson explains why he wrote the first edition (2003) and the reason for the second edition.

In each chapter, Stephenson provides code segments or complete code and shows the resulting image. Many of these images are included in the color plates, with some extra images illustrating the power of RenderMan. Most chapters have a summary of the idea and include related commands. The book is divided into three parts: overview, geometry, and shading.

The overview includes what RenderMan is and isn't; a summary of the RenderMan API features; where to get software that implements the API; and how to use the book. I urge you to follow the author's suggestions.

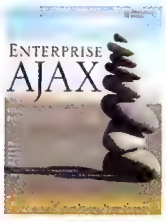
The geometry section uses mostly spheres, although the standard teapot makes an appearance. Stephenson starts with a simple scene, extends it by using positioning commands, and then adds other round objects, such as cones. He then adds coloring options to the code and introduces camera and lighting features to enhance the final image. At this point, the surface is plain, which leads readers to the next four chapters, which address standard and complex surface types, shadows with shadow maps, motion blur, and depth of field. Stephenson shows the C language interface. I wondered why, until he used it to enhance an image with multiple objects and features (difficult to accomplish by hand coding).

The shading section deals with writing code for a variety of shaders, starting with the basic one introduced in part 2. The chapters cover, in order: simple lighting; color ramps; simple, tiling, and repeating patterns; projections and coordinate spaces, including the math details for those interested; painted textures; displacements; noise; aliasing and how it solves many, but not all, problems; highlights models; advanced lighting; and, finally, GI (global illumination). Stephenson points out the advantages and disadvantages of the various methods and shows ways to improve the final image with code samples.

I found the book to be a good introduction to the subject, although a bit terse in spots. On the flip side, Stephenson refers to a related Web site, which I could not find listed in the book, and the last three chapters have some misspelled words (not a lot, but enough to show a certain lack of care). — *Howard Whitston*

Enterprise Ajax: Strategies for Building High-performance Web Applications

David Johnson, Alexei White, Andre Charland, Prentice Hall PTR, 2007, $39.99, ISBN: 0132242060.



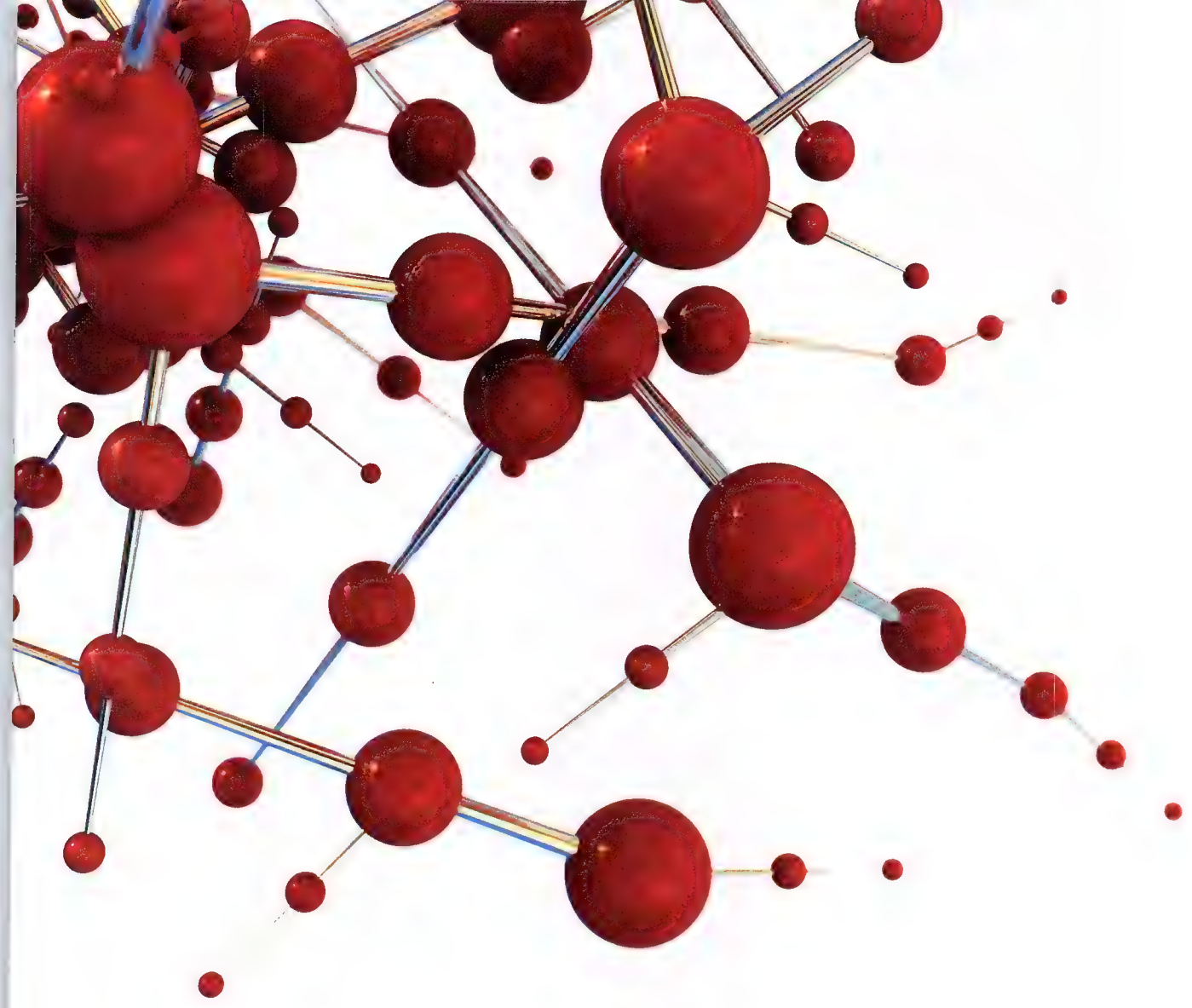
The authors of *Enterprise Ajax* explain why Ajax (Asynchronous JavaScript and XML) offers such great promise in large-scale development. The book was written for advanced enterprise developers, architects, and user interface designers, covering the key concepts and techniques of Ajax as a large-scale development methodology. The authors also systematically introduce today's key Ajax techniques and components.

The best features of the book include: its in-depth coverage of the use of Ajax to implement MVC (model-view-controller) in the browser; how to encapsulate user interface functionality; how to avoid the security challenges associated with Ajax Web applications; and Ajax usability improvement, including the use of the back button, caching, bookmarking, and offline usage.

The Web site that complements the book (http://www.enterpriseajax.com) contains code samples, case studies, tutorials, and demos. This is an essential feature for a book that addresses such a dynamic topic as Ajax.

Chapter 5 offers something new to the Ajax literature. It examines the Ajax development life cycle, identifying what is specific for Ajax applications. This chapter is important to project leaders who need to build a development framework and overcome the performance and security problems associated with Ajax.

This book is a necessary addition to the library of anyone interested in the development of robust enterprise Web applications. — *Jose M. Ramirez*



APRIL

MySQL Conference
April 14-17, 2008
Santa Clara, California
http://en.oreilly.com/mysql2008/public/content/home

LEET 08 (Usenix Workshop on Large-Scale Exploits and Emergent Threats)
April 15, 2008
San Francisco, California
http://www.usenix.org/events/leet08/

Software Test and Performance Conference
April 15-17, 2008
San Mateo, California
http://www.stpcon.com/

Open Publish Conference
April 23-24, 2008
Baltimore, Maryland
http://www.open-conferences.com/baltimore/

Open Standards 2008: Composability within SOA Symposium
April 28-May 1, 2008
Santa Clara, California
http://events.oasis-open.org/home/symposium/2008

MAY

Web Design World
May 5-7, 2008
Chicago, Illinois
http://webdesignworld.com/2008/chicago/

Star*East*
May 5-9, 2008
Orlando, Florida
http://www.sqe.com/StarEast/

Ruby on Rails Bootcamp
May 5-9, 2008
Atlanta, Georgia
http://bignerdranch.com/classes/ruby.shtml

JavaOne
May 6-9, 2008
San Francisco, California
http://java.sun.com/javaone/sf/index.jsp

Where 2.0 Conference
May 12-14, 2008
Burlingame, California
http://en.oreilly.com/where2008/public/content/home

VSLive!
May 12-16, 2008
Orlando, Florida
http://vslive.com/2008/orlando/

Online Game Development Conference
May 13-15, 2008
Seattle, Washington
http://www.mpogd.com/news/?ID=3001

JUNE

Gartner IT Security Summit
June 2-4, 2008
Washington, D.C.
http://gartner.com/it/page.jsp?id=594029

Microsoft Tech-Ed Developers
June 3-6, 2008
Orlando, Florida
http://www.microsoft.com/events/teched2008/developer/default.mspx

Better Software Conference
June 9-12, 2008
Las Vegas, Nevada
http://www.sqe.com/BetterSoftwareConf/

Python Bootcamp
June 9-13, 2008
Atlanta, Georgia
http://bignerdranch.com/classes/python.shtml

Usenix Annual Technical Conference
June 22-27, 2008
Boston, Massachusetts
http://www.usenix.org/events/usenix08/

O'Reilly Velocity: Web Performance and Operations Conference
June 23-24, 2008
Burlingame, California
http://en.oreilly.com/velocity2008/public/content/home

JULY

Ubuntu Live
July 21-22, 2008
Portland, Oregon
http://en.oreilly.com/ubuntu2008/public/content/home

O'Reilly OSCON
(Open Source Convention)
July 21-25, 2008
Portland, Oregon
http://en.oreilly.com/oscon2008/public/content/home

To announce an event, E-MAIL
QUEUE-ED@ACM.ORG OR
FAX +1-212-944-1318

Continued from page 72
tautologically, just the ones that caused harm? And harm to whom? We can read beyond the title that Dijkstra is carefully "hedging his bets." No computer is worth its chips without a branch-on-negative operator. So, it's not so much the "going-to" another distant part of your program that's harmful, it's how to write readable control jumps in higher-level languages. GOTO-less programming means cleaner, structured code; GOTO-full programs were messy and hard to maintain, producing what the critics called spaghetti code, an insult to all pasta lovers.[4]

Many language theologians defend the compound GOTO known as the CASE statement. I go further by claiming God's approval. No peer review can be peerer! Nature's own Grand Unified Law is but a set of better-than-realtime nested CASE statements. I've seen the listings. Alas, I was allowed the skimpiest walk-through, and that only after signing a terrifying nondisclosure thingy of Faustian proportions. Dare I risk revealing the gist? Well, just entre nous. Note that God uses C++ (no surprise there, Bjarne):

```
#include <DixieDean.h> // divine headers, q.g. [quod
google]

T = -tPlanck; // start clock just ahead of time

switch BigBang {
        case 0: wait (tPlanck); t-break; ...
        case 1: switch word {
        ...
        case googol^googol: { ... }
        ...
```

As far as I could check with HyperLint, every eventuality seems to be covered without a single exception throw needed. Readers reading this on April 1 are warned that I'm aware that I seem to be presenting a discredited, deterministic, discretely countable universe. Not necessarily. The symbols 0, 1, and googol are not to be interpreted by human C++ standards. God gave us the integers, and God can override the integers.

Dijkstra, "spreading his bets," suggests another helpful idiom, leading to game theory and risk analysis. For here we reach real life as it is lived by the vast majority of my readers. A moment of truth approaches, demanding honest introspection. If your beliefs have no discernible impact on your behavior, can we really believe the sincerity of your belief-claims? Russell was fond of testing someone's belief that it was going to rain. Did that someone stay indoors or venture out without her brolly? Sooner or later, balances must be balanced, weights weighed, decisions made, bets placed, acts enacted. But how to pick the best (or least-worst) hypothesis?

Occam's razor just shaves off a few baby-hairs from rival hypotheses. Solomon's sword divides the whole baby, offering each mother an evenly balanced slice. But hurry before someone empties the baby with the bathwater. (Philosophers love these confusing homely parables.) We nod knowingly over the Biblical tale, admiring Solomon's wisdom in identifying the biological mother. Don Watson applies modern jurisprudence to the case with telling results ("Solomon Reversed on Appeal")[5], while I'm inclined to ponder the outcome if the mothers had been equally versed in game theory brinkmanship. Calling them Mrs. A and Mrs. B, we must assume that each sincerely believes in her claimed maternal relationship. Each must weigh the credibility of Solomon's threat to cleave the babe in twain. And each must assign values to the possible outcomes: one mother winning full custody or neither; both gaining gruesome half-custody, which can be taken as at least denying the rival mother custody. Twisting the aphorism, "It's not enough to succeed, others must fail," we reach this: "It's not so bad losing as long as others do not win." Jim Waldo's courage is in great demand.

In our own fair trade, extreme, opinionated, backbiting positions are taken on the most fundamental issues, such as "What is or are data, and how should it or they be processed?" As that curmudgeon's curmudgeon, Arthur Schopenhauer, meant to say, "Opinions are like bums, everybody's got one."[6]

Even from our 60 glorious ACM years of relatively polite ruminations, the rumble of language and operating-system religious wars breaks through. Open Software's Open Wounds? Methodological Madness? The Association of Computing Machinations? And up for grabs on the ground floor (awaiting the $1 million P = NP Clay Institute Prize): Completely Unprovable Incompleteness? As we quipped in the FastRandy Univac days, "I hear the sound of distant drums!" But let's not exaggerate.

FROM BOOLE TO BOOTLE

It's hard, not to say pointless, to formalize this balance of extremes. But Booles rush in.[7] Extending my great, grand (meaning famously renowned) Great-Grand-Uncle George's binary logic, we have the SKB-quasi-continuous-ternary system with +N (true), -N (false), iN (meaningless). Here, N is an integer >= 0 intended, somehow, to indicate plausibility or evidential-support; *i* is either

of the square roots of minus 1. Your choice, if you think it matters. The aim is to "quantify" propositions over an integral Gaussian lattice (q.g.).

T(P) = (x, y) = x + iy, the truth-value of P, is meant to indicate some mix of truth, falsehood, and meaninglessness. With y = 0 (giving T(P) real values), we are close to traditional binary Booleans but with x > 0 meaning true, x < 0 meaning false, and with the added nuance that x1 > x2 > 0 means that x1 is more true than the true x2. Similarly 0 > x1 > x2 means that x2 is falser than the false x1. Reassuringly, all truths, however small, are truer than all falsehoods, however small! For x = 0, y !=0, we conveniently locate all truly meaningless propositions on the y-axis. I'm working on a meaningful metric for meaningless, one that meets our daily intuition that some propositions are clearly dafter than others. Thus, y1 > y2 means y1 is more meaningless than y2. The rules for logical implication must also be carefully extended beyond the traditional (p -> q) = (!p v q). We must retain the old truth tables for real values of T(P), so that true never implies false, but I've never been happy with false implies true or false. We would certainly hope to prevent "daft" implying anything "dafter," while "daft" can safely imply "less daft." Formally, T(0, y1) -> T(0, y2) for all y1 >= y2 > 0. But T(0, y1) !-> T(0, y2) for all y2 >= y1 > 0. I call this the Limiting Insanity Clause in honor of the Marx Brothers.

The singularity at the origin T(P) = (0, 0) is reserved for genuine, meaningful don't-knows. Don't push me on this. Epistemologist Donald Rumsfeld has not returned my calls. Some think they know that the truths or falsehoods of Goldberg's conjecture (P = NP) are presently unknown but in essence knowable. That is, one day, before the stars lose their glory,[8] we'll nail them as true or false. Others think they may be in essence undecidable a la Goedel. One thing is certain. I'll never get a Fields medal unless those ageist bastards increase the MaxAge >>> 40.

THE TEXT IS TAKEN FROM...

High on my list of why-bothers is a recent "translation" by the Australian Bible Society of the Bible into mobile-phone text-speak format. This SMS Bible is ever-so-predictably unclever when u fnd @ da strt of Gnsis: "In da Bginnin God cre8d da heavens & da earth. Da earth waz barren, wit no 4m of life..." Surely not blasphemous as some sensitive souls have screamed. And you can hardly say that anything is literally lost in translation, since the SMS is almost an automatic token-by-token mapping of the earlier CEV (Contemporary English Version) from the American Bible Society. En route via untold redactions from the original Hebrew, Aramaic, and Greek via Tyndall to the KJV (King James Version and its gender-free Ruler James Version) and on to the CEV, there are paths that many bemoan as undignified dumbing-downs. Is the SMS version just a gimmicky last straw? Or will hoodies dig the message and see the light?

Will this weak rebic[9] file-compression scheme ever find a place among all our Zip and archiving tools? I say Stuffit! Incidentally, TTY has reemerged as the abbreviation for this telephone-text mode. Olde Fartes will recall the ancient peripheral as defined in my *Devil's DP Dictionary* (McGraw-Hill, 1981):

TTY n. \pronounced 'titty'\ [Acronym for TeleTYpe] Any terminal of the teletype vintage in which the restricted character set is more than offset by the unique busy signal, viz., printer noise. See also GLASS TTY.

To save you hunting in your attic for the tattered remains [sic] of my book, here's the cross-ref:

GLASS TTY n. \pronounced 'glass titty'\ A CRT terminal so lacking in features that it behaves like a TTY.

CRT n. [Cathode Ray Tube] Originally an important storage device, developed by Prof. F.C. Williams, Manchester University, in 1947, but now relegated to trivial applications in the timesharing and entertainment environments. See also GLASS TTY.

At this rate I could exhaust you and the dictionary tracking overt and covert links. I'll leave it for now with the oft-quoted daddy of all cross-references:

recursive adj. See recursive.

Thanks for the reader feedback on conflicting aphorism-pairs. I'll report the winners in my next column. Q

REFERENCES

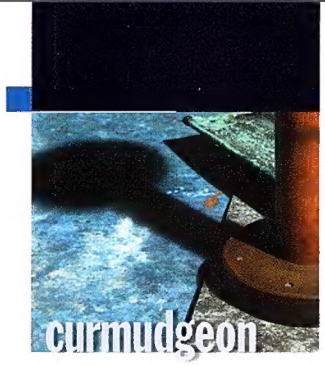
1. *Understating* is not the use of footnotes, which are a few of my favorite things, along with Julie Andrews, the sound of music, and positive reader feedback. Note, too, that *understanding* really means *overstanding*.
2. Scouse mothers' malapropisms are called *malapudlianisms*. See "Lern Yerself Scouse," Stan Kelly, Fritz Spiegl, Frank Shaw (Scouse Press, 1966).
3. "GOTO Statement Considered Harmful." Letter to *Communications of the ACM*. March 1968; reprinted in the 60th anniversary issue.

4. Older readers will know of R. Lawrence Clark's response to Dijskstra's letter: the infamous COME FROM construct (*Datamation*, December 1973). What started as a spoof became a serious joke involving programs running backwards; http://www.fortran.com/come_from.html.
5. Watson, D. E. 1992. Solomon Reversed on Appeal; http://www.enformy.com/$solomon.html.
6. For some genuine Schopenhauerian quotes and insights, see *The Consolations of Philosophy*, Alain de Botton, Penguin, 2001. Schopenhauer's own opinions were even stronger than Nabokov's or Dijkstra's. Thus: "If only I could get rid of the illusion of regarding the generation of vipers and toads as my equals, it would be a great help to me" (op. cit., 176).
7. The Bootle-Boole family connection has now been verified at Disney's GenomeLand. The T-gene mutation can be dated to the Irish potato famine, compounded with spelling errors by a Lincoln Registrar.
8. You'll share my relief that cosmological eschatologists have extended the cosmic lifespan to 300 trillion years. Make them all count! Start running that NP program NOW.
9. Readers are invited to coin a more precise word than *rebus* for the simpler set of phonograms used in text-speak (1 = won; 4 = for; 6 = sex; 8 = ate; Gr86 = great sex; G! = gangbang; more). No doubt the new mobile phones, as pocket servers/workstations/satnavs/tv studios, can handle the full range of graphic rebuses.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

STAN KELLY-BOOTLE (http://www.feniks.com/skb/; http://www.sarcheck.com), born in Liverpool, England, read pure mathematics at Cambridge in the 1950s before tackling the impurities of computer science on the pioneering EDSAC I. His many books include *The Devil's DP Dictionary* (McGraw-Hill, 1981), *Understanding Unix* (Sybex, 1994), and the recent e-book *Computer Language—The Stan Kelly-Bootle Reader. Software Development Magazine* has named him as the first recipient of the new annual Stan Kelly-Bootle Eclectech Award for his "lifetime achievements in technology and letters." Neither Nobel nor Turing achieved such prized eponymous recognition. Under his nom-de-folk, Stan Kelly, he has enjoyed a parallel career as a singer and songwriter. He can be reached at curmudgeon@acmqueue.com.



Solomon's Sword Beats Occam's Razor

Stan Kelly-Bootle, Author

Choosing YOUR BEST HYPOTHESIS

I've told you a googol times or more: Don't exaggerate! And, less often, I've ever-so-gently urged you not to understate.[1] Why is my advice ignored? Why can't you get IT... just right, balanced beyond dispute? Lez Joosts Mildews, as my mam was fond of sayin,[2] boxing both my ears with equal devotion. Follow the Middle Way as Tao did in his Middle Kingdom. Or "straight down the middle," as golfer Bing Crosby used to croon. His other golf song was "The Wearing of the Green," but such digressions run counter to my straight, plow-on-ahead advice. I've just smoked a cigarette branded Cleopatra, but that's none of your beeswax neither, and strictly between me and my Egyptian placements sponsor.

So, shun deviations and avoid life's bunkers lurking left and right. Our current presidential candidates excel in this craftiness, being both pro-Nafta and anti-Nafta as the local polls dictate. Yet, by one of those many quirks of natural language, politicians seeking "compromises" often find their reputations "compromised."

In pondering the attributes that make for good, well-shod scientists, and in particular, good systems designers and developers, an intuitive sense of balance looms large. We work from incomplete specifications and ill-defined, oft-conflicting aims. Do you want IT *now* or do you want IT *functional*? There are few finer discussions of these challenges than Jim Waldo's 2004 essay, "On System Design" (available via http://portal.acm.org/citation.cfm?id=1167533).

Waldo pondered anew why things were getting worse. After reviewing the "design" process, and reminding us of the term's ambiguity (systems may exhibit *design* without having been *designed*), he traces the environmental causes for this decline, such as intellectual property constraints and poor training. There's always some element of "us-them" scapegoating when things go wrong: echoes of Hardy wrongly blaming Laurel, which always niggles me because "Another fine mess you've gotten us into, Stan" easily triggers my guilty self-defensive reactions. (Why are there so few Stans around, whether Stanleys or Stanislauses?)

Waldo, unlike the whinging Hardy, suggests real remedies of the agile and open persuasions to reduce the mess, but without excluding the occasions when heavy and closed subterfuge is needed. Interestingly, he stresses the need for courage at the grass-roots, nonmanagerial level—for example, in the ways masters should teach their crafts and delegate to apprentices. At the same time, Waldo urges solutions that don't require impractical major revolutions in the managerial infrastructure. You need to work around existing organizational hazards. Thus, we return to my opening theme of balancing between opposites. This is not the same as cowardly (and possibly painfully) sitting on fences. We envisage more a hidden door or two in the fence accessible to the qualified gnostic who is free to roam unhindered, admiring the views in many fields.

We face the apparent paradox that it's possible and useful to honestly hold mutually contradictory beliefs simultaneously. The Greek *dilemma* for two such opposing beliefs can be extended to *trilemma* for three, and on to *n-lemma*, but we strongly recommend small positive integral values for *n*. We can exclude Quineans whose heads are buzzing with propositions and quoted "propositions" that mean their opposites: that way madness lies, and a possible collapse of meaningful meanings. What I have in mind (to coin a phrase!) is hinted at by the idiom "playing both sides down the middle." There are plausible beliefs, which we can poshly call hypotheses, of the type Karl Popper called "falsifiable." Indeed, he used falsifiability to distinguish scientific hypotheses from those that may well be meaningful but somehow fall outside the nit-probing methods of science. Trying to ignore the sniff of circularity (which is all around us on such occasions), we seek ways of distinguishing between beliefs that require endless individual observational verifications to sustain them and beliefs that, at least conceptually, could admit to a sudden single deflating counter-example (see this column, "Some Swans are Black," July/August 2007).

Precision in wording one's beliefs is paramount, of course, when moving between formal and informal statements. Did Dijkstra consider all GOTOs harmful[3] or,

Continued on page 69